WU
WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

EFMD
EQUIS
ACCREDITED

ePub^WU Institutional Repository

Claudio Di Ciccio and Fabrizio Maria Maggi and Marco Montali and Jan Mendling

Resolving inconsistencies and redundancies in declarative process models

Article (Submitted)

http://epub.wu.ac.at/

Resolving Inconsistencies and Redundancies in Declarative Process Models

Claudio Di Ciccio[a,*], Fabrizio Maria Maggi[b], Marco Montali[c],
Jan Mendling[a]

[a]*Vienna University of Economics and Business, Austria*
[b]*University of Tartu, Estonia*
[c]*Free University of Bozen-Bolzano, Italy*

Abstract

Declarative process models define the behaviour of business processes as a set of constraints. Declarative process discovery aims at inferring such constraints from event logs. Existing discovery techniques verify the satisfaction of candidate constraints over the log, but completely neglect their interactions. As a result, the inferred constraints can be mutually contradicting and their interplay may lead to an inconsistent process model that does not accept any trace. In such a case, the output turns out to be unusable for enactment, simulation or verification purposes. In addition, the discovered model contains, in general, redundancies that are due to complex interactions of several constraints and that cannot be cured using existing pruning approaches. We address these problems by proposing a technique that automatically resolves conflicts within the discovered models and is more powerful than existing pruning techniques to eliminate redundancies. First, we formally define the problems of constraint redundancy and conflict resolution. Second, we introduce techniques based on the notion of automata-product monoid, which guarantees the consistency of the discovered models and, at

---

*Corresponding author.
E-mail address: claudio.di.ciccio@wu.ac.at
Postal address: Vienna University of Economics and Business, Institute for Information Business (Building D2, Entrance C) – Welthandelsplatz 1, A-1020 Vienna, Austria
Phone number: +43 1 31336 5222

Email addresses: `claudio.di.ciccio@wu.ac.at` (Claudio Di Ciccio),
`f.m.maggi@ut.ee` (Fabrizio Maria Maggi), `montali@inf.unibz.it` (Marco Montali),
`jan.mendling@wu.ac.at` (Jan Mendling)

the same time, keeps the most interesting constraints in the pruned set. The level of interestingness is dictated by user-specified prioritisation criteria. We evaluate the devised techniques on a set of real-world event logs.

1. Introduction

The automated discovery of processes is the branch of the process mining discipline that aims at constructing a process model on the basis of the information reported in event data. The underlying assumption is that the recorded events indicate the sequential execution of the to-be-discovered process activities. The compact and correct representation of the behaviour observed in event data is one of the major concerns of process mining. Process discovery algorithms are classified according to the type of process model that they return, i.e., either procedural or declarative. Procedural process discovery techniques return models that explicitly describe all the possible executions allowed by the process from the beginning to the end. The output of declarative process discovery algorithms consists of a set of constraints, which exert conditions on the enactment of the process activities. The possible executions are implicitly established as all those ones that respect the given constraints. Mutual strengths and weaknesses of declarative and procedural models are discussed in [1, 2].

One of the advantages of procedural models such as Petri nets is the rich set of formal analysis techniques available. These techniques can, for instance, identify redundancy in terms of implicit places or inconsistencies like deadlocks [3]. In turn, similar facilities are not provided for novel declarative modelling languages like DECLARE. This is a problem for several reasons. First, we are currently not able to check the consistency of a generated constraint set. Many algorithms that generate DECLARE models include in the output those constraints that are individually satisfied in the log in more than a given number of cases. The interaction of returned constraints is thereby neglected, with the consequence that subsets of constraints can end up contradicting one another. Second, it is currently unclear whether a given constraint set is free of redundancies. Since there are constraint types that imply one another, it is possible that the generated constraint sets are partially redundant. The lack of formal techniques for handling these two issues

2

is unsatisfactory from both a research and a practical angle. This is also a roadblock for conducting fair comparisons in user experiments when a Petri net without deadlocks and implicit places is compared with a constraint set of unknown consistency and redundancy-freedom.

In this paper, we address the need for formal analysis of DECLARE models. We define the notion of an *automata-product monoid* as a formal notion for analysing consistency and local minimality, which is grounded in automata multiplication. Based on this structure, we devise efficient analysis techniques. Our formal concepts have been implemented as part of a process mining tool that we use for our evaluation. By analysing event log benchmarks, we are able to show that inconsistencies and redundancies occur in process models automatically discovered by state-of-the-art tools. First, our technique can take such process models as input and return constraints sets that are consistent. To this end, contradictory subsets are identified and resolved by removing the constraints generating the conflict. Second, our technique eliminates those constraints that do not restrict the behaviour of the process any further, i.e., that do not convey any meaningful information to the output. As a consequence, the returned sets are substantially smaller than the ones provided by prior algorithms, though keeping the expressed behaviour equivalent to the inconsistency-free process. This paper extends the research presented in our former publication [4] with a complete and self-consistent definition of the adopted formal concepts and algorithms. We also provide alternative strategies to be utilised during the redundancy and consistency check, so as to allow for different criteria to prioritise the constraints during the pruning phase. This is of crucial importance, since manipulating a declarative process model towards removal of inconsistencies and redundancies is intrinsically expensive from a computational point of view. Furthermore, we introduce a complementary technique to further reduce the number of redundancies in the models after the first check. Finally, we broadly extend the evaluation with an analysis of our implemented approach over real-world data sets including the event logs provided for the former editions of the BPI challenge.

The paper is structured as follows. Section 2 illustrates intuitively the problems that we tackle with the proposed research work. Section 3 describes the preliminary notions needed to formally contextualise the challenged issues. Section 4 formally specifies the problems of inconsistencies and redundancies in detail. Section 5 defines our formal notion of automata-product monoid, which offers the basis to formalise the techniques for consistency

and redundancy checking. Section 6 illustrates the results of our evaluations based on real-world benchmarking data. Section 7 discusses our contributions in the light of related work. Finally, Section 8 concludes the paper.

2. Motivation

Declarative process models consist of sets of constraints exerted on tasks, which define the rules to be respected during the process execution. A well-established language for modelling declarative processes is DECLARE [5, 6]. DECLARE defines a set of default *templates*, which are behavioural rules that refer to parameters in order to abstract from tasks. In DECLARE, e.g., $Init(x)$ is a template imposing that a given parametric task x must be the one with which every process instance starts. $End(x)$ specifies that every process instance must terminate with the given task x. $Response(x, y)$ states that if task x is carried out, then task y must be eventually executed afterwards. $Precedence(x, y)$ imposes that y can only be performed if x has been previously executed.

Let us consider a simple example process having three tasks, a, b, and c. By indicating the execution sequence of tasks with their name, possible enactments that fulfil a process model consisting of $Init(a)$ and $End(c)$ are: (i) ababab, and (ii) ababac. If we consider an event log made of the aforementioned execution sequences and use any declarative discovery algorithm to reveal a declarative process model that could have generated them, it would correctly return a set of constraints including $Init(a)$ and $End(c)$ because they are always satisfied. However, the set of constraints would include also (1) $Precedence(a, b)$ and (2) $Precedence(a, c)$, as well as (3) $Response(a, c)$ and (4) $Response(b, c)$: Those four constraints hold true in the event log as well. Nevertheless, if a is already bound to be the first task to be carried out in every process instance ($Init(a)$), clearly no other task can be executed if a is not done before. Therefore, the first two constraints can be trivially deduced by $Init(a)$. They add no information, yet they contribute to uselessly enlarge the set of constraints returned to the user as the outcome of the discovery. By the same line of reasoning, the third and fourth constraints are superfluous with respect to $End(b)$. Intuitively, this example outlines the problem of *redundancy*, which is one of the two challenges that we tackle with this research work: The objective is to remove from the set of constraints in the discovered process model those ones that do not add information, i.e., that

4

are not restricting the process behaviour any further given the remaining ones.

In the context of declarative process discovery, event logs can be affected by recording errors or report exceptional deviations from the usual enactments [7]. In such cases, constraints that were originally part of the process may be violated in some of the recorded executions. If discovery algorithms take into account only those constraints that always hold true in the event log, a minimum amount of noise might already cause several constraints to be discarded from the returned set [8, 9, 10]. To circumvent this issue, declarative discovery algorithms offer the possibility to tune a so-called *support threshold*: It specifies the minimum fraction of cases in which a constraint is fulfilled within the event log to let such constraint be included in the discovered model. However, this comes at the price of possibly having conflicts in the model though: Constraints that hold true in a fraction of the event log above the set threshold can contradict other constraints. In such a case, the model becomes unsatisfiable, i.e., it exerts conditions that cannot be met by any possible execution. Such a model would clearly be to no avail to the discovery intents. This issue outlines the problem of *inconsistencies* in the discovered model, which we challenge in this research paper.

The aim of the presented approach is therefore twofold: Given a discovered declarative process model, we want to (1) remove its inconsistencies, and (2) remove its redundancies. To pursue these objectives, we aim at keeping the process behaviour as similar as possible to the original one when removing inconsistencies, and retaining the minimum number of constraints that still represent the same original behaviour while getting rid of the redundancies. The number of combinations of constraints to test for the optimum of both problems is not tractable in practice, because every subset of the original constraints set should be confronted with the others. Our solution instead requires a polynomial number of checks over constraints to provide a sub-optimal yet effective solution. Furthermore, different criteria can be adopted to express (1) the desired behavioural closeness and (2) the preferability of constraints to be retained. To this extent, our solution envisages (1) the relaxation of conditions exerted by the contradicting constraints and (2) different ranking criteria for constraints, respectively.

3. Declarative process modelling and mining

This section defines the formal background for our research problem. In particular, we introduce and revisit the concepts of event logs and of declarative process modelling and mining.

Notational conventions. We adopt the following notations. Given a set X, *(i)* the multi-set of X is denoted as $\mathbb{M}(X)$; *(ii)* the power-set of X is denoted as $\mathbb{P}(X)$; *(iii)* a sequence of elements $x_i \in X$ is denoted by the juxtaposition of its elements $x_1 x_2 \cdots x_n$; *(iv)* the cardinality of X is denoted as $|X|$; the same notation applies both to the length of sequences and the cardinality of multi-sets.

Identifiers in cursive sans-serif format will be written in sans-serif letters when assigned to actual parameters. Generic identifiers of tasks, e.g., will be indicated as a, b, and c, whereas concrete assignments of task identifiers will be written as a, b, and c.

3.1. Event Logs

An *event* is a system-recorded information reporting on the execution of a task during the execution of a process. Events are labelled with so-called *event classes*, i.e., task names [11]. We assume that each event uniquely corresponds to the execution of a single task. This assumption builds upon the work on event class reconciliation of Baier et al. [12, 13]. We thus abstract every event with its event class, in turn related to a task. A finite sequence of events is named *trace*. A complete trace represents the execution of a process instance from the beginning to the end. An *event log* is a collection of traces. If multiple process instances have been executed by conducting the same sequence of tasks, multiple traces in the event log consist of the same sequence of events, accordingly.

Formally, an *event log* L is a multi-set of traces t_i with $i \in [1, |L|]$, which in turn are finite sequences of events $e_{i,j}$ with $i \in [1, |L|]$ and $j \in [1, |t_i|]$ [14, 15, 16]. The *log alphabet* $\mathcal{A}$ is the set of symbols identifying all possible tasks and event classes. We write a, b, c to refer to them. Without loss of generality, we also refer to events as occurrences of symbols in $\mathcal{A}$. By denoting the set of sequences of tasks as $\mathcal{A}^*$ we have that $L \in \mathbb{M}(\mathcal{A}^*)$. An example of log is: $\mathsf{L} = \{\mathsf{t_1}, \mathsf{t_2}\}$ where $\mathsf{t_1} = \mathsf{abcacbacd}$ and $\mathsf{t_2} = \mathsf{ababc}$. Event logs are the fundamental input of automated process discovery algorithms [14, 17].

Type	Notation	Template and description	Regular expression
Existence templates		**Cardinality templates**	
	$\boxed{\overset{1.*}{x}}$	$Participation(x)$ x occurs at least *once*	`[^x]*(x[^x]*)+[^x]*`
	$\boxed{\overset{0.1}{x}}$	$AtMostOne(x)$ x occurs at most *once*	`[^x]*(x)?[^x]*`
		Position templates	
	$\boxed{\overset{INIT}{x}}$	$Init(x)$ x is the *first* to occur	`x.*`
	$\boxed{\overset{END}{x}}$	$End(x)$ x is the *last* to occur	`.*x`
Relation templates		**Forward-unidirectional relation templates**	
	$\boxed{x}\!\!-\!\!\bullet\!\!-\!\!\boxed{y}$	$RespondedExistence(x, y)$ If x occurs, then y occurs too	`[^x]*((x.*y.*)|(y.*x.*))*[^x]*`
	$\boxed{x}\!\!\bullet\!\!\rightarrow\!\!\boxed{y}$	$Response(x, y)$ If x occurs, then y occurs after x	`[^x]*(x.*y)*[^x]*`
	$\boxed{x}\!\!\bullet\!\!\Rightarrow\!\!\boxed{y}$	$AlternateResponse(x, y)$ If x occurs, y occurs afterwards before x recurs	`[^x]*(x[^x]*y[^x]*)*[^x]*`
	$\boxed{x}\!\!\bullet\!\!\Rrightarrow\!\!\boxed{y}$	$ChainResponse(x, y)$ If x occurs, y occurs immediately after it	`[^x]*(xy[^x]*)*[^x]*`
		Backward-unidirectional relation templates	
	$\boxed{x}\!\!\rightarrow\!\!\bullet\!\!\boxed{y}$	$Precedence(x, y)$ y occurs only if preceded by x	`[^y]*(x.*y)*[^y]*`
	$\boxed{x}\!\!\Rightarrow\!\!\bullet\!\!\boxed{y}$	$AlternatePrecedence(x, y)$ y occurs only if preceded by x with no other y in between	`[^y]*(x[^y]*y[^y]*)*[^y]*`
	$\boxed{x}\!\!\Rrightarrow\!\!\bullet\!\!\boxed{y}$	$ChainPrecedence(x, y)$ y occurs only if x occurs immediately before it	`[^y]*(xy[^y]*)*[^y]*`
		Coupling templates	
	$\boxed{x}\!\!\bullet\!\!-\!\!\bullet\!\!\boxed{y}$	$CoExistence(x, y)$ x occurs iff. y occurs	`[^xy]*((x.*y.*)|(y.*x.*))*[^xy]*`
	$\boxed{x}\!\!\bullet\!\!\rightarrow\!\!\bullet\!\!\boxed{y}$	$Succession(x, y)$ x occurs iff. it is followed by y	`[^xy]*(x.*y)*[^xy]*`
	$\boxed{x}\!\!\bullet\!\!\Rightarrow\!\!\bullet\!\!\boxed{y}$	$AlternateSuccession(x, y)$ x and y occur iff. they follow one another, alternating	`[^xy]*(x[^xy]*y[^xy]*)*[^xy]*`
	$\boxed{x}\!\!\bullet\!\!\Rrightarrow\!\!\bullet\!\!\boxed{y}$	$ChainSuccession(x, y)$ x and y occur iff. y immediately follows x	`[^xy]*(xy[^xy]*)*[^xy]*`
		Negative templates	
	$\boxed{x}\!\!\bullet\!\!\Rrightarrow\!\!\!\!|\!\!\bullet\!\!\boxed{y}$	$NotChainSuccession(x, y)$ x and y occur iff. y does not immediately follow x	`[^x]*(aa*[^xy][^x]*)*([^x]*|x)`
	$\boxed{x}\!\!\bullet\!\!\rightarrow\!\!\!|\!\!\bullet\!\!\boxed{y}$	$NotSuccession(x, y)$ x can never occur before y	`[^x]*(x[^y]*)*[^xy]*`
	$\boxed{x}\!\!\bullet\!\!-\!\!|\!\!-\!\!\bullet\!\!\boxed{y}$	$NotCoExistence(x, y)$ x and y never co-occur	`[^xy]*((x[^y]*)|(y[^x]*))?`

<div align="center">Table 1: DECLARE templates.</div>

3.2. Declarative Process Modelling Languages

A declarative process modelling language represents the behaviour of processes by means of constraints, i.e., rules that must not be violated during the execution of process instances. Such rules are meant to be exerted over tasks in the context of temporal structures like imperative process models or logs. To date, DECLARE is one of the most well-established declarative process modelling languages. It provides a standard library of templates (*repertoire*), i.e., behavioural constraints parametrised over activities. Table 1 lists the constraints that will be considered in this paper. Typical examples of DECLARE constraints are *Participation*(a) and *Response*(b, c). The former specifies that a must be executed in every process instance. The latter declares that if b is executed, then c must eventually follow. The constrained task of *Participation*(a) is a, whereas the constrained tasks of *Response*(b, c) are b and c. The template of *Participation*(a) is *Participation*, whilst the template of *Response*(b, c) is *Response*. *Participation* is an example of existence template, because it asserts conditions on the execution of a single activity. *Response* is an example of a relation template as it specifies conditions over pairs of activities. For relation templates, *activations* and *targets* are defined: The former is a task whose execution imposes obligations on the enactment of another task, i.e., the target. E.g., b is the activation and c is the target of *Response*(b, c), because the execution of b requires c to be executed eventually.

Formally, a template is a predicate $\mathcal{C}/_n \in \mathfrak{C}$, where $\mathfrak{C}$ is the DECLARE repertoire and n denotes its arity, i.e., the number of parameters [18]. In this article, we consider constraints of arity not higher than 2, because they constitute the subset of constraints discovered by the majority of declarative process miners [19, 20, 21]. Nevertheless, the presented approach can be seamlessly extended to the case of constraints of higher arity. Existence templates are unary, whereas relation templates are binary. Formal parameters of constraints are denoted by x and y for binary constraints or as $x_1, \ldots, x_n$ for constraints of a generic arity n. We will interchangeably use, e.g., $\mathcal{C}/_2$ or $\mathcal{C}(x, y)$ to denote a template of arity 2. Templates will be denoted as $\mathcal{C}$ when their arity is unspecified or clear from the context.

A constraint is the application of a template over tasks by means of the assignment of its formal parameters to elements in $\mathcal{A}$. Formally, given a template $\mathcal{C}/_n \in \mathfrak{C}$, a *parameter assignment* γ_n is a function $[1, n] \to \mathcal{A}$, where $[1, n]$ is the set of integers ranging from 1 to n. $\gamma_n(i)$ is meant to assign the i-th parameter of $\mathcal{C}/_n$ to a task in $\mathcal{A}$, in compliance with the positional

notation of the parameters of predicates. The $\mathcal{C}/_n$-constraint resulting from γ_n, written $\mathcal{C}\gamma/_n$, is also represented as $C = \mathcal{C}(\gamma_n(1), \ldots, \gamma_n(n))$. For example, $Response(\mathsf{b}, \mathsf{c})$ denotes the constraint resulting from the application of $\gamma_2 : \{1, 2\} \to \mathcal{A}$ to the $Response/_2$ template, where $\gamma_2(1) = \mathsf{b}$ and $\gamma_2(2) = \mathsf{c}$.

In light of the above, and taking inspiration from the tabular representation of behavioural relations in [22, 23], we can define a declarative process model as follows.

Definition 3.1 (Declarative process model). A declarative process model is a tuple $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ where:

- $\mathcal{A}$ is a finite non-empty set of tasks;

- $\mathfrak{C}$ is a finite non-empty repertoire of templates;

- $\Gamma = \{\mathcal{C}\gamma/_{n_i} : \mathcal{C}/_n \in \mathfrak{C}, \quad \gamma_{n_i} : [1, \ldots, n] \to \mathcal{A}, \quad i \geqslant 0\}$ is a finite set of constraints.

Γ is a subset of the constraints universe $\mathfrak{C}^{\mathcal{A}}$, which corresponds to the set of all constraints C that derive from the instantiation of every template $\mathcal{C}/_n \in \mathfrak{C}$ with every possible assignment to tasks in $\mathcal{A}$. Please notice that by definition two different assignments γ_n and γ'_n can be applied to the same constraint $\mathcal{C}/_n$. It is the case, e.g, when $Response(\mathsf{a}, \mathsf{b})$ and $Response(\mathsf{b}, \mathsf{c})$ both belong to the declarative process model: $\gamma_2(1) = \mathsf{a}$ and $\gamma_2(2) = \mathsf{b}$, $\gamma'_2(1) = \mathsf{b}$ and $\gamma'_2(2) = \mathsf{c}$, hence $Response\gamma/_2 = Response(\mathsf{a}, \mathsf{b})$ and $Response\gamma'/_2 = Response(\mathsf{b}, \mathsf{c})$.

DECLARE is a declarative process modelling language providing the repertoire of templates listed in Table 1. Having, e.g., $\mathsf{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, in DECLARE $\mathfrak{C}^{\mathsf{A}}$ would contain $Init(\mathsf{a})$, $Init(\mathsf{b})$, $Init(\mathsf{c})$, $Response(\mathsf{a}, \mathsf{b})$, $Response(\mathsf{b}, \mathsf{a})$, $Response(\mathsf{b}, \mathsf{c})$, $Response(\mathsf{c}, \mathsf{b})$, $Response(\mathsf{a}, \mathsf{c})$, $Response(\mathsf{c}, \mathsf{a})$, etc.

We respectively indicate activation and target of a template $\mathcal{C}$ as $\mathcal{C}|_{\bullet}$ and $\mathcal{C}|_{\Rightarrow}$. Hence, $Response(x, y)|_{\bullet} = x$, and $Response(x, y)|_{\Rightarrow} = y$. With a slight abuse of notation, we use the same notation also for constraints: $Response(\mathsf{a}, \mathsf{b})|_{\bullet} = \mathsf{a}$, $Response(\mathsf{a}, \mathsf{b})|_{\Rightarrow} = \mathsf{b}$. Moreover, we assume that for a template $\mathcal{C}(x)$ of arity 1, activation and target coincide: $\mathcal{C}(x)|_{\bullet} = \mathcal{C}(x)|_{\Rightarrow} = x$.

As said, events are meant to be recordings of the tasks carried out during the process enactment. Therefore, we will interchangeably interpret DECLARE rules as *(i)* behavioural relations between tasks in a process model or *(ii)* conditions exerted on the occurrence of events in traces. We will henceforth consider that, e.g., $Participation(\mathsf{a})$ imposes that every trace contains at least an occurrence of a. Likewise, $Response(\mathsf{b}, \mathsf{c})$ indicates that after

the occurrence of b, c occurs afterwards in the trace. Both t_1 and t_2 in the example log L are compliant with *Participation*(a) and *Response*(b, c). Such conceptual matching is typical of DECLARE mining approaches [9, 24], as event logs are used to analyze to what extent constraints are respected by counting the number of fulfilments within traces.

3.2.1. Evaluation and Satisfiability of a Declarative Process Model

Since constraints are predicates, they can be *evaluated* and checked for *satisfiability*. In particular, as exposed by Räim et al. [25], every constraint of a declarative process model can be evaluated over traces by adopting a semantics based on linear temporal structures. We thus introduce the notion of evaluation of a constraint over a trace as a function $\eta : \mathfrak{C}^{\mathcal{A}} \times \mathcal{A}^* \to \{\top, \bot\}$, such that:

$$\eta\left(C, t\right) = \begin{cases} \top & \text{if } C \in \mathfrak{C}^{\mathcal{A}} \text{ evaluates to true over } t \in \mathcal{A}^* \\ \bot & \text{otherwise.} \end{cases} \tag{1}$$

If the conditions imposed by the constraint are satisfied by every event in the trace, then the trace fulfils the constraint, i.e., the constraint evaluates to true ($\top$) over the trace.

With a slight abuse of notation, we denote the evaluation of a declarative process model $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ over a trace by means of the same symbol η, and define it as follows:

$$\eta\left(\mathcal{M}, t\right) = \begin{cases} \top & \text{if for all } C \in \Gamma \subseteq \mathfrak{C}^{\mathcal{A}}, \ C \text{ evaluates to true over } t \in \mathcal{A}^* \\ \bot & \text{otherwise.} \end{cases}$$
$$\tag{2}$$

The notion of evaluation of a declarative process model leads to the *satisfiability* problem, i.e., checking whether there exists a trace over which the model evaluates to true. Hereinafter, we denote the set of traces that satisfy a declarative process model $\mathcal{M}$ as its *language*:

$$\mathscr{L}(\mathcal{M}) = \{t \in \mathbb{P}\left(\mathcal{A}^*\right) : \eta\left(\mathcal{M}, t\right) = \top\}. \tag{3}$$

Since a declarative process model is evaluated to $\top$ on a trace t only if *all* its constraints are evaluated to $\top$ on t (Equation (2)), given two models $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ and $\mathcal{M}' = \langle \mathcal{A}, \mathfrak{C}, \Gamma' \rangle$ where $\Gamma' \subseteq \Gamma$, it follows that $\mathscr{L}(\mathcal{M}) \subseteq \mathscr{L}(\mathcal{M}')$. The declarative process model $\mathcal{M}_I = \langle \mathcal{A}, \mathfrak{C}, \emptyset \rangle$

trivially accepts any trace consisting of elements of $\mathcal{A}$, because it imposes no constraints. Hence, $\mathscr{L}(\mathcal{M}_I) = \mathbb{P}(\mathcal{A}^*)$ is the most underfitting declarative process model, but not interesting from a business process perspective, since it accepts any possible behaviour. Likewise, *empty* process models, i.e., not satisfiable by any trace, are equivalently ineffective. As an example, we can consider the DECLARE constraints $CoExistence(\mathsf{a}, \mathsf{b})$ and $NotCoExistence(\mathsf{a}, \mathsf{b})$. The first one states that in a trace a and b always co-occur. The second one establishes that if a occurs in a trace, then b cannot occur, and vice-versa. It follows that a model $\mathcal{M}_\emptyset = \langle \{\mathsf{a}, \mathsf{b}\}, \mathfrak{C}, \{CoExistence(\mathsf{a}, \mathsf{b}), NotCoExistence(\mathsf{a}, \mathsf{b})\} \rangle$ is unsatisfiable, because the two constraints are contradicting, which implies $\mathscr{L}(\mathcal{M}_\emptyset) = \emptyset$. Notice that also the declarative process model $\mathcal{M}_\emptyset' = \langle \{\mathsf{a}, \mathsf{b}\}, \mathfrak{C}, \{CoExistence(\mathsf{a}, \mathsf{b}), NotCoExistence(\mathsf{a}, \mathsf{b}), Init(\mathsf{a})\} \rangle$ is still unsatisfiable, although $Init(\mathsf{a})$ is not in contradiction with $CoExistence(\mathsf{a}, \mathsf{b})$ nor with $NotCoExistence(\mathsf{a}, \mathsf{b})$. This observation leads to the problem that we want to address: Finding and removing contradicting constraints that make declarative process models unsatisfiable.

3.2.2. Discovery of a Declarative Process Model

A declarative process model can be *discovered* by evaluating all constraints in the constraints universe over the event log and returning all and only those constraints that evaluate to $\top$ over the event log. However, this would make the discovered model overfitting, with the consequence that if the event log contained errors, then the discovered model would be affected by erroneously discarded or added constraints [26]. To overcome this issue, metrics have been introduced that make the discovered model less prone to faulty log entries.

Taking inspiration from the area of data mining [27], we adopt the *support* metric [19, 21]. Support assesses the degree of fulfilment of constraints in the event log by scaling the number of traces fulfilling the constraint by the number of traces in the log. Support is defined as a function $\sigma : \mathfrak{C}^\mathcal{A} \times \mathbb{M}(\mathcal{A}^*) \to [0, 1] \subseteq \mathbb{R}_{\geqslant 0}$, being $\mathbb{R}_{\geqslant 0}$ the set of positive real numbers, computed as follows:

$$\sigma(C, L) = \frac{|\{t \in L : \eta(C, t) = \top\}|}{|L|}. \tag{4}$$

Given the example event log from above, $\mathsf{L} = \{t_1, t_2\}$ where $t_1 = \mathsf{abcacbacd}$ and $t_2 = \mathsf{ababc}$, we have that: $\sigma(Participation(\mathsf{a}), \mathsf{L}) = 1.0$,

$\sigma(Response(\mathsf{b},\mathsf{c}),\mathsf{L}) = 1.0$, and $\sigma(Response(\mathsf{a},\mathsf{b}),\mathsf{L}) = 0.5$, because no b follows the last a occurring in t_1.

Typically, a discovered declarative process model consists of those constraints having a support higher than a user-specified threshold: Those that are fulfilled in a significant number of cases belong to the discovered model. However, the amount of constraints that the discovered model consists of is usually overwhelming, when only relying on such criterion. Therefore, metrics for assessing the relevance of constraints have been established, i.e., *confidence* κ and *interest factor* ι, which scale the support by the ratio of traces in which the activation occurs, resp. both the constrained tasks occur. Confidence and interest factor are defined as functions $\kappa : \mathfrak{C}^{\mathcal{A}} \times \mathbb{M}(\mathcal{A}^*) \to [0,1] \subseteq \mathbb{R}_{\geqslant 0}$, and $\iota : \mathfrak{C}^{\mathcal{A}} \times \mathbb{M}(\mathcal{A}^*) \to [0,1] \subseteq \mathbb{R}_{\geqslant 0}$, respectively, which are computed as follows:

$$\kappa(C,L) = \sigma(C,L) \times \frac{|\{t \in L : C|_{\bullet} \in t\}|}{|L|}, \tag{5}$$

$$\iota(C,L) = \sigma(C,L) \times \frac{|\{t \in L : C|_{\bullet} \in t \text{ and } C|_{\Rightarrow} \in t\}|}{|L|}. \tag{6}$$

Different variants of calculating these metrics have been proposed [19, 28, 29]. Notice that both κ and ι scale the value of σ by a number included in the range $[0,1]$. By their definition, it always holds true that given a constraint C and an event log L, $0 \leqslant \iota(C,L) \leqslant \kappa(C,L) \leqslant \sigma(C,L) \leqslant 1$.

3.3. Declare Template Types and Subsumption

DECLARE is a declarative process modelling language that provides a repertoire of templates for the specification of constraints over tasks. The list of templates considered in this paper is provided in Table 1. Here, we describe how the templates are divided into *types* and constitute a subsumption hierarchy [30, 28, 26], as illustrated in Figure 1.

Examples of constraints in DECLARE are: *(i) Participation(a)*, specifying that task a must occur in every trace; *(ii) AtMostOne(a)*, declaring that a must occur not more than once in a trace; *(iii) RespondedExistence(a,b)*, imposing that if a occurs in a trace, then also b must occur in the same trace. *Participation* and *AtMostOne* are existence templates. Because they exert restrictions on the number of occurrences of a task in a trace, they belong to the type of *cardinality constraint templates*. *Init(a)* and *End(a)* are existence constraints stating that a must be the first, resp. the last, event occurring in

(a) Existence templates
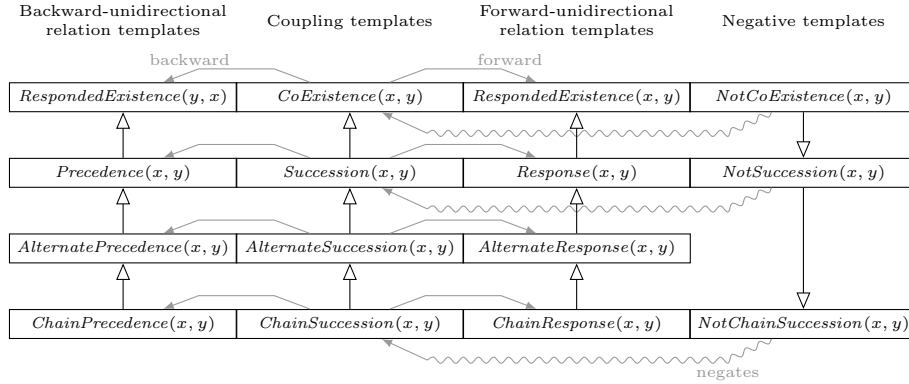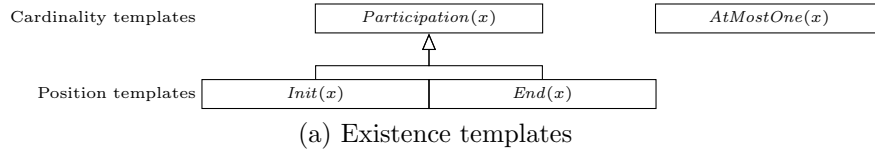

(b) Relation templates

Figure 1: The subsumption map of DECLARE templates. Templates are indicated by solid boxes. The subsumption relation is depicted as a line starting from the subsumed template and ending in the subsuming one, with an empty triangular arrow recalling the UML IS-A graphical notation. The coupling constraint templates are linked to the related forward-unidirectional relation constraint and backward-unidirectional relation constraint templates by means of grey arcs. The negative constraint templates are graphically linked to the corresponding coupling constraint templates by means of wavy grey arcs.

all traces. Therefore, they belong to the type of *position constraints*. Both their templates are subsumed by *Participation*, because they imply that the constrained task occurs in every trace in order to be the first or the last one. Figure 1(a) illustrates the subsumption hierarchy of existence constraint templates. Templates are indicated in solid boxes. The subsumption between templates is drawn with a line starting from the subsumed template and ending in the subsuming one, with an empty triangular arrow recalling the UML IS-A graphical notation.

As Figure 1(b) illustrates, *RespondedExistence*(x, y) generates an offspring of related relation templates. Its directly subsumed templates ("children") are *Response*(x, y) and *Precedence*(y, x). *Response*(a, b) imposes that eventually after an occurrence of a (the activation), b (the target) must occur. Dually, *Precedence*(a, b) requires that before an occurrence of the activation task b, target task a occurs. Both constraints strengthen the conditions

13

exerted by *RespondedExistence* by specifying that not only must the target occur, but also in which relative position in the trace (after or before the activation). However, the role of activation and target are swapped in *Precedence*, w.r.t. *RespondedExistence*. Owing to this, *RespondedExistence* and *Response* belong to the type of *forward-unidirectional relation* templates, whereas *Precedence* is a *backward-unidirectional relation* template.

Technically, given two n-ary templates $\mathcal{C}, \mathcal{C}' \in \mathfrak{C}$, we say that $\mathcal{C}$ *is subsumed by* $\mathcal{C}'$, written $\mathcal{C} \sqsubseteq \mathcal{C}'$, if for every trace $t \in \mathcal{A}^*$ and every parameter assignment γ_n from the parameters of $\mathcal{C}$ to tasks in $\mathcal{A}$, whenever t complies with the instantiation of $\mathcal{C}$ with γ_n, then t also complies with the instantiation of $\mathcal{C}'$ with γ_n. For binary templates, we write $\mathcal{C} \sqsubseteq \mathcal{C}'^-$ if the subsumption holds by inverting the parameters of $\mathcal{C}'$ w.r.t. those in $\mathcal{C}$, i.e., by considering templates $\mathcal{C}(x,y)$ and $\mathcal{C}'(y,x)$. We thus have that $Response \sqsubseteq RespondedExistence$. By the same line of reasoning, we have that $Precedence \sqsubseteq RespondedExistence^-$. $\sqsubseteq$ is transitive (if $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C}' \sqsubseteq \mathcal{C}''$ then $\mathcal{C} \sqsubseteq \mathcal{C}''$) and reflexive ($\mathcal{C} \sqsubseteq \mathcal{C}$). We also introduce the inverse relation of $\sqsubseteq$, i.e., $\sqsupseteq$, which we use to indicate that $\mathcal{C}$ *subsumes* $\mathcal{C}'$, i.e., $\mathcal{C} \sqsupseteq \mathcal{C}'$: E.g., $RespondedExistence \sqsupseteq Response$, and $RespondedExistence \sqsupseteq Precedence^-$. In the following, we extend the usage of $\sqsubseteq$ and its inverse relation to constraints too: $C \sqsubseteq C'$ means that constraint C is subsumed by constraint C', e.g., $Response(\mathsf{a}, \mathsf{b}) \sqsubseteq RespondedExistence(\mathsf{a}, \mathsf{b})$.

In the subsumption hierarchy of both *Response* and *Precedence*, the direct child templates are *AlternateResponse* and *AlternatePrecedence*. The concept of alternation strengthens the parent template by adding the condition that between pairs of activation and target, not any other activation occurs. The subsumption hierarchy concludes with *ChainResponse* and *ChainPrecedence*: They impose that occurrences of activation and target are immediately adjacent.

The conjunction of a forward-unidirectional relation template and a backward-unidirectional relation template belonging to the same level of the subsumption hierarchy generates the so-called coupling templates: $Succession(x, y)$, e.g., holds when both $Response(x, y)$ and $Precedence(x, y)$ hold true. In addition, the coupling template *CoExistence* is equal to the conjunction of *RespondedExistence* and *RespondedExistence*$^-$. For every coupling template $\mathcal{C}$, a function $fw(\mathcal{C})$ and $bw(\mathcal{C})$ are defined that resp. return the related forward-unidirectional relation and the backward-unidirectional relation templates. Hence, $fw(Succession(x, y)) = Response(x, y)$ and $bw(Succession(x, y)) = Precedence(x, y)$. In Figure 1(b), the functions fw

14

and *bw* are indicated by grey arcs labelled as *forward* and *backward*, respectively. With a slight abuse of notation, we will adopt function symbols *fw* and *bw* not only for templates but also for constraints.

Finally, coupling templates *CoExistence*, *Succession* and *ChainSuccession* correspond to other templates that share the same activations and exert opposite conditions on the targets: Resp., *NotCoExistence*, *NotSuccession* and *NotChainSuccession*. For instance, *CoExistence*(a, b) states that a and b always co-occur in a trace. *NotCoExistence*(a, b) states instead that if either a (resp. b) occurs in the trace, then b (resp. a) cannot. Owing to this, *NotCoExistence*, *NotSuccession* and *NotChainSuccession* are named negative templates. Given a negative template, e.g., *NotCoExistence*, we say that it *negates* the corresponding coupling template, e.g., *CoExistence*. Due to the opposite conditions exerted on the targets, the subsumption hierarchy gets also reverted w.r.t. the corresponding negated templates: *NotCoExistence* $\sqsubseteq$ *NotSuccession* $\sqsubseteq$ *NotChainSuccession*. In Figure 1(b), negative templates are graphically linked to their corresponding coupling templates by means of wavy grey arcs labelled as *negates*.

Based on the concept of subsumption, we can define the notion of relaxation $\mathscr{R}$. $\mathscr{R}$ is a unary operator that returns the direct parent in the subsumption hierarchy of a given template. If there exists no parent for the given template, then $\mathscr{R}$ returns a predicate that would hold true for any possible trace, i.e., $\top$. Formally, given a template $\mathcal{C} \in \mathfrak{C}$, we have:

$$\mathscr{R}(\mathcal{C}) = \begin{cases} \mathcal{C}' & \text{if } \textit{(i)} \ \mathcal{C}' \in \mathfrak{C} \setminus \{\mathcal{C}\}, \textit{(ii)} \ \mathcal{C} \sqsubseteq \mathcal{C}', \text{ and} \\ & \quad \textit{(iii)} \ \nexists \mathcal{C}'' \in \mathfrak{C} \setminus \{\mathcal{C}, \mathcal{C}'\} \text{ s.t. } \mathcal{C} \sqsubseteq \mathcal{C}'' \text{ and } \mathcal{C}'' \sqsubseteq \mathcal{C}' \\ \top & \text{otherwise} \end{cases} \quad (7)$$

We extend the relaxation operator and the subsumption relation to the domain of constraints, such that, e.g., $\mathscr{R}(Response(\mathsf{a}, \mathsf{b})) = RespondedExistence(\mathsf{a}, \mathsf{b})$.

3.4. Semantics of DECLARE as regular expressions

A plethora of semantics for DECLARE templates have been proposed in the literature by relying on different logic-based approaches, including Linear Temporal Logic (LTL) [31, 32], Linear Temporal Logic on Finite Traces (LTL$_f$) [33, 34], First Order Logic (FOL) formulae over finite ordered traces [28, 34] and abductive logic programming [35]. We adopt regular expressions (REs) because they allow us to take advantage of well-established techniques

15

for calculating automata products, which are at the base of our approach. Table 1 lists the translation of templates into REs.

Regular expressions are a formal notation to compactly express finite sequences of characters, a.k.a. matching strings. The syntax of REs consists of any juxtaposition of characters of a given alphabet, optionally grouped by enclosing parentheses (and), to which the following operators can be applied: Binary alternation | and concatenation, and the unary Kleene star *. Thus, the regular expression `a(bc)*d|e` identifies any string starting with `a`, followed by any number of repetitions of the pattern (sub-string) `bc` (optionally, none) and closed by either `d` or `e`, such as `ad`, `abcd`, `abcbce` and `ae`. Table 1 adopts the POSIX standard for the following additional shortcut notations: *(i)* `.` and `[^x]` respectively denote any character or any character but x, *(ii)* `+` and `?` operators respectively match from one to any and from none to one occurrences of the preceding pattern. We also make use of *(iii)* the parametric quantifier `{,m}`, with `m` integer higher than 0, which specifies the maximum number of repetitions of the preceding pattern, and *(iv)* the parametric quantifier `{n,}`, with `n` integer higher than or equal to 0, which specifies the minimum number of repetitions of the preceding pattern. We recall here that *(i)* REs are closed under the conjunction operation & [36], and *(ii)* the expressive power of REs completely covers regular languages [37], thus *(iii)* since regular grammars are recognisable through REs [38, 37], for every RE, a corresponding deterministic finite state automaton (FSA) exists, accepting all and only the matching strings [39]. The conjunction operator & satisfies commutativity and associativity. Its identity element is `.*`.

3.5. Finite State Automata

A (deterministic) FSA is a finite-state labelled transition system $A = \langle \Sigma, S, s_0, \delta, S_f \rangle$, where: Σ is an alphabet; S is the finite non-empty set of states; $s_0 \in S$ is the initial state; $\delta : S \times \Sigma \to S$ is the transition function, i.e., a function that, given a starting state and a character of the alphabet, returns the target state (if defined); $S_f \subseteq S$ is the set of final (accepting) states [38]. For the sake of simplicity, we will omit the qualification "deterministic". A finite path π of length n over A is a sequence $\pi = \langle \pi^1, \ldots, \pi^n \rangle$ of tuples $\pi^i = \langle s^{i-1}, \sigma^i, s^i \rangle \in \delta$, for which the following conditions hold true: *(i)* π^1, the first tuple, is such that $s^0 = s_0$ (i.e., π starts from the initial state of A) and *(ii)* the starting state of π^i is the target state of π^{i-1}: $\pi = \langle \langle s^0, \sigma^1, s^1 \rangle, \langle s^1, \sigma^2, s^2 \rangle, \ldots \langle s^{n-1}, \sigma^n, s^n \rangle \rangle$. A finite string of length $n \geqslant 0$, i.e., a concatenation $\sigma = \sigma_1 \ldots \sigma_n$ of characters $\sigma_i \in \Sigma$, is

16

(a) Identity element   (b) Absorbing element

Figure 2: Finite state automata acting as identity element and absorbing element for the automata cross-product operation.

accepted by A if a path π of length n is defined over A and is such that *(i)* for every $i \in [1, n]$, $\pi^i = \langle s^{i-1}, \sigma_i, s^i \rangle$, and *(ii)* $\pi^n = \langle s^{i-1}, \sigma_n, s^n \rangle$ is s.t. $s^n \in S_{\mathrm{f}}$. We overload the $\mathscr{L}$ notation by denoting as $\mathscr{L}(A) \subseteq \mathbb{P}(\Sigma^*)$ the (possibly infinite) set of strings accepted by A.

FSAs are closed under the product operator $\times$ [40]. A product of two FSAs A and A' accepts the intersection of languages (sets of accepted strings) of each operand: $\mathscr{L}(A \times A') = \mathscr{L}(A) \bigcap \mathscr{L}(A')$. The product of FSAs is an isomorphism for the conjunction of REs, i.e., the product of two FSAs respectively corresponding to two REs is equivalent to the FSA that derives from the conjunction of the two REs [41]: Given the REs r, r', and naming as $\mathscr{A}$ the operation leading from an RE to the corresponding FSA, we have that $\mathscr{A}(r \& r') = \mathscr{A}(r) \times \mathscr{A}(r')$. The product operator $\times$ is commutative and associative. The identity element for $\times$ over the alphabet Σ is $A^I = \langle \Sigma, \{s_0\}, s_0, \{s_0\} \times \Sigma \times \{s_0\}, \{s_0\} \rangle$ (Figure 2(a)). It accepts all strings over Σ: $\mathscr{L}(A^I) = \mathbb{P}(\Sigma^*)$. The absorbing element is $A^\emptyset = \langle \Sigma, \{s_0\}, s_0, \emptyset, \emptyset \rangle$ (Figure 2(b)). It does not accept any string: $\mathscr{L}(A^\emptyset) = \emptyset$.

4. Formalisation of the Problem

In this section, we present the twofold problem tackled in this work. First, we want to avoid that the discovered declarative process models contain inconsistencies, i.e., contradictions among constraints that make the overall model unsatisfiable. Second, we want to minimise the number of constraints in the discovered declarative process models, in particular by eliminating those that are redudant.

4.1. The Consistency Problem

In Section 3.2, we have introduced the general notions of declarative process model and of its language, defined in terms of the set of traces that satisfy all constraints present in the model. We have also discussed that not

all declarative process models are meaningful. One extreme case is the one in which the declarative process $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ of interest is unsatisfiable, i.e., $\mathscr{L}(\mathcal{M}) = \emptyset$. In this case, $\mathcal{M}$ cannot be used for simulation nor execution, since there exists no trace that satisfies it. In addition, the usage of $\mathcal{M}$ to evaluate the compliance of a log confuses the process analyst, since every trace is trivially considered non-compliant. Furthermore, $\mathcal{M}$ acts as an absorbing element when composing it with another declarative model $\mathcal{M}'$, in the sense that the model resulting from the composition continues to be unsatisfiable, irrespectively of the constraints contained in $\mathcal{M}'$.

An unsatisfiable model $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ contains at least one constraint $\overline{C} \in \Gamma$ that is in *conflict* with the other constraints $C_1, \ldots, C_{|\Gamma|-1} \in \Gamma \setminus \{\overline{C}\}$, i.e., for which no trace exists that satisfies them all. Formally, there is no $t \in \mathcal{A}^*$ such that $\eta(C, t) = \top$ for every $C \in \Gamma \setminus \{\overline{C}\}$ and $\eta(\overline{C}, t) = \top$. Addressing the *consistency problem* means ensuring that a declarative process model is satisfiable, i.e., accepts at least one execution trace. When the process model is unsatisfiable, this requires to identify and remove those constraints that are in conflict.

This problem is extremely challenging. In fact, there could be multiple sets of conflicting constraints, each formed by two or more constraint. Their identification is thus inherently intractable, as it requires in the worst case to consider all possible subsets of Γ [42]. Furthermore, once such conflicting sets are singled out, there are in general exponentially many ways of removing constraints belonging to such sets so as to fix the inconsistency. This issue is particularly difficult to manage in the context of declarative process discovery, since each newly discovered constraint could suddenly introduce a conflict with the partial declarative process model discovered so far.

On the other hand, the consistency problem is pervasive in declarative process discovery. To illustrate this, we utilise the event log set of the BPI challenge 2012 [43]. The event log pertains to an application process for personal loans or overdrafts of a Dutch bank. It contains 262,200 events distributed across 24 different possible event classes and 13,087 traces. Process mining tools such as MINERful [28] and Declare Maps Miner [19] generate declarative process models in DECLARE from event logs. In essence, these models define a set of declarative constraints that collectively determine the allowed and the forbidden traces.

The main idea of declarative process discovery is that the overfitting of the discovered models can be avoided by defining thresholds for parameters such as support, confidence and interest factor. By choosing a support

threshold smaller than 100%, we can easily obtain constraint sets that are supported by different parts of the log and that contradicts each other. E.g., when using MINERful on the BPI challenge 2012 event log with a support threshold of 75%, it returns the constraints $Participation(\mathsf{A\_Preaccepted})$, $NotChainSuccession(\mathsf{A\_Preaccepted}, \mathsf{W\_Completeren\ aanvrag})$, and $ChainResponse(\mathsf{A\_Preaccepted}, \mathsf{W\_Completeren\ aanvrag})$, which have an empty set of traces that fulfil all of them. In fact, the first constraint imposes that $\mathsf{A\_Preaccepted}$ must be executed at least once, the second constraint imposes that $\mathsf{A\_Preaccepted}$ is never directly followed by $\mathsf{W\_Completeren\ aanvrag}$, whereas the third one requires that if $\mathsf{A\_Preaccepted}$ is executed, $\mathsf{W\_Completeren\ aanvrag}$ must immediately follow. Clearly, such inconsistent constraint sets should not be returned by the discovery algorithm.

4.2. The Minimality Problem

The second problem we tackle in this work is minimality. This problem is concerned with the informative content of the discovered declarative process model. The goal is to understand whether all its constraints effectively contribute to the separation between compliant and non-compliant traces, or are instead redundant. Let $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ and let $C \in \Gamma$ be a constraint of $\mathcal{M}$. Intuitively, we say that C is redundant in $\mathcal{M}$ if the set of compliant traces (i.e., the language defined by $\mathcal{M}$) is not affected by the presence of C. Formally, let $\mathcal{M}' = \langle \mathcal{A}, \mathfrak{C}, \Gamma \setminus \{C\} \rangle$ be the declarative process model obtained from $\mathcal{M}$ by removing constraint C. We then have that C is redundant in $\mathcal{M}$ if $\mathscr{L}(\mathcal{M}) = \mathscr{L}(\mathcal{M}')$.

In this light, addressing the minimality problem means transforming the discovered declarative process model into one that is language-wise equivalent, but does not contain redundant constraints. Models that contain redundancies are pointlessly difficult to understand for the process analysts, since redundant constraints do not provide any additional information about the permitted and forbidden behaviours.

Like for consistency, this problem is inherently difficult and calls for the application of suitable strategies that find a reasonable trade-off between optimality and computational efficiency. In fact there are, in general, exponentially many ways of making a model redundancy-free. This is, again, particularly critical in the context of declarative process discovery. Every newly discovered constraint could in fact introduce redundancy, which could be removed either by ignoring the newly discovered constraint or by dropping a set of already discovered constraints redundant with the new one.

To show to which extent this problem is present in concrete declarative process modelling languages such as DECLARE, we recall the fact that DECLARE templates can be organised in a hierarchy of constraints, depending on a notion of subsumption, as discussed in Section 3.3. This notion of subsumption is tightly related to that of redundancy, since a constraint can be immediately recognised as redundant if it is subsumed by another constraint present in the partial declarative process model discovered so far. However, we stress the fact that redundancy could be detected, in some cases, only by analysing the model as a whole, and not just considering pairs of constraints.

When using MINERful on the BPI challenge 2012 event log with a support threshold of 75%, it returns the constraints $ChainResponse(\mathsf{A\_Submitted}, \mathsf{A\_PartySubmitted})$ and $NotChainSuccession(\mathsf{A\_Submitted}, \mathsf{A\_Accepted})$. The latter constraint is clearly redundant, because the former requires the first task following $\mathsf{A\_Submitted}$ to be $\mathsf{A\_PartySubmitted}$. Therefore, no other task but $\mathsf{A\_PartySubmitted}$ can directly follow. A fortiori, $\mathsf{A\_Submitted}$ and $\mathsf{A\_Accepted}$ cannot be in direct succession. Clearly, such redundant constraint pairs should not be returned.

4.3. Framing the Problem

In Section 4.1 and Section 4.2, we have introduced the issues of consistency and redundancy in declarative process models. We now frame these problems in the context of declarative process discovery.

Our goal is to define *effective* post-processing techniques that, given a previously discovered DECLARE model $\mathcal{M}$ possibly containing inconsistencies and redundancies, manipulate it by removing inconsistencies and reducing redundancies, but still retaining as much as possible its original structure. In this respect, the post-processing is completely agnostic to the process mining algorithm used to generate the model as well as to the input event log.

This latter assumption makes it impossible to understand how much a variant of the discovered model fits with the log. However, we can at least assume that each single constraint in $\mathcal{M}$ retains the support, confidence, and interest factor that were calculated during the discovery phase. These values can be used to decide which constraints have to be prioritised, and ultimately decide whether a variant $\mathcal{M}'$ of $\mathcal{M}$ has to be preferred over another variant $\mathcal{M}''$.

In principle, we could obtain an optimal solution by exhaustive enumeration, executing the following steps:

1. The vocabulary $\mathcal{A}$ of $\mathcal{M}$ is extracted;

2. The set $\mathfrak{C}^{\mathcal{A}}$ of all possible candidate constraints is built;

3. The power-set $\mathbb{P}\left(\mathfrak{C}^{\mathcal{A}}\right)$ of all possible subsets of $\mathfrak{C}^{\mathcal{A}}$, i.e., of all possible DECLARE models using constraints in $\mathfrak{C}^{\mathcal{A}}$, is computed;

4. A set of candidate models $\widehat{\mathcal{M}}$ over $\mathcal{A}$ and $\mathfrak{C}$ is obtained from $\mathbb{P}\left(\mathfrak{C}^{\mathcal{A}}\right)$, by filtering away those models that are inconsistent or contain redundant constraints;

5. A ranking of the models in $\widehat{\mathcal{M}}$ is established, considering their similarity to the original discovered model $\mathcal{M}$.

However, this exhaustive enumeration is unfeasible in the general case, given the fact that it requires to iterate over the exponentially many models in $\mathbb{P}\left(\mathfrak{C}^{\mathcal{A}}\right)$, an intractably huge state space. Consequently, we devise a heuristic algorithm that mediates between optimality of the solution and computational efficiency. In summary, its main features are the following:

- It produces as output a consistent variant of the initial model $\mathcal{M}$. This is a strict, necessary requirement.

- The algorithm works in an incremental fashion, i.e., it constructs the variant of $\mathcal{M}$ by iteratively selecting constraints. Once a constraint is added, it is not retracted from the model. This is done by iterating through the candidate constraints in descending order of suitability. The degree of suitability is dictated by an ordering relation that sorts the constraints before the algorithm starts the checking phase. An example of such an ordering relation is the ranking of constraints on the basis of their support, confidence, and interest factor: It makes the algorithm retain the constraints that better fit the log from which they were discovered. On the one hand, this drives our algorithm to favour more suitable constraints and remove less suitable constraints in the case of an inconsistency or a redundancy. On the other hand, this has a positive effect on performance and also guarantees that the algorithm is deterministic.

- Due to incrementality, the algorithm is not guaranteed to produce a final variant that is redundancy-free and minimal in the number of constraints, but we still achieve a local minimum. Our experimental findings show that this local minimum is satisfactory, since the algorithm is
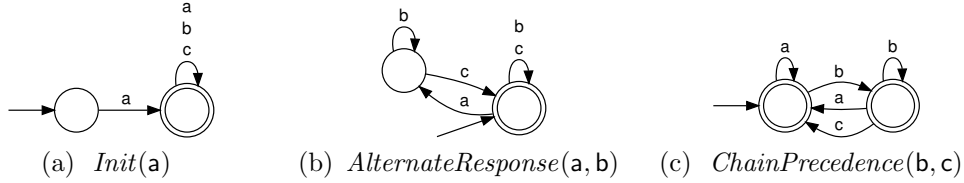
| (a) *Init*(a) | (b) *AlternateResponse*(a, b) | (c) *ChainPrecedence*(b, c) |

Figure 3: FSAs accepting the traces compliant with some Declare constraints over the log alphabet {a, b, c}.

able to significantly reduce the number of redundant constraints w.r.t. the state-of-the-art discovery algorithms.

5. The approach

This section describes how we tackle the problem of finding a non-redundant consistent DECLARE model in a way that reduces the intractable theoretical complexity. First, we present the algebraic structure on top of which the check of redundancies and conflicts is performed: It bases upon the mapping of the conjunction of DECLARE constraints to the product of FSAs. Thereafter, we define and discuss the algorithm that allows us to pursue our objective. In particular, we rely on the associativity of the product of FSAs. This property allows us to check every constraint one at a time and include it in a temporary solution. This is done by saving the product of the constraints checked so far with the current one. For the selection of the next candidate constraint to check, we make use of a greedy heuristic, which explores the search space by gathering at every step the constraint that has the highest support or is most likely to imply the highest number of other constraints. Notice that the commutativity of the automata product guarantees that conflicting constraints are found, regardless of the order with which they are checked. The algorithm proceeds without visiting the same node in the search space twice.

5.1. Constraints as Automata

As already shown in [21], DECLARE constraints can be formulated as regular expressions (REs) over the log alphabet. The assumption is that every task in the log alphabet is bi-univocally identified by a character. Thus, traces can be assimilated to finite sequences of characters (i.e., strings) and

(a) Product automaton of *Init*(a) and *AlternateResponse*(a, b)

(b) Product automaton of *Init*(a), *AlternateResponse*(a, b) and *ChainPrecedence*(b, c)
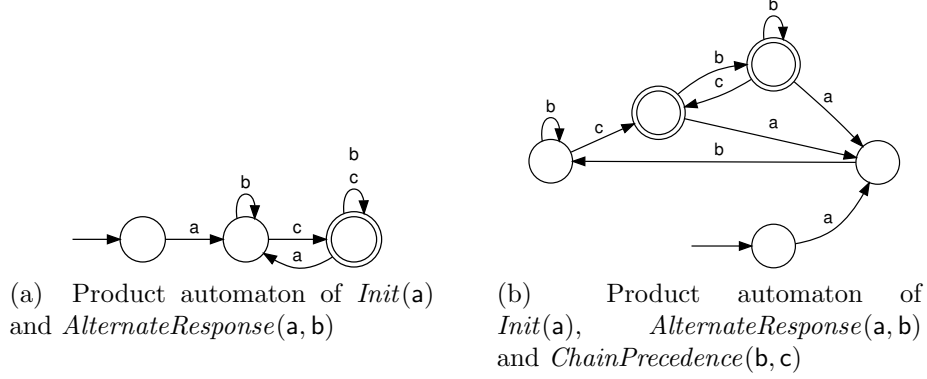
Figure 4: Product automata of the FSAs shown in Figure 3

regular languages represent the traces allowed by a DECLARE model. A constraint is thus evaluated to true over a trace if and only if the corresponding string is matched by the constraint's regular expression.

Using the POSIX wildcards, we can express, e.g., *Init*(a) as `a.*`, and *Response*(a, b) as `[^a]*(a.*b)*[^a]*`. The comprehensive list of transpositions for DECLARE templates is listed in Table 1 and explained in [41]. Henceforth, we will refer to such a mapping as $\mathscr{E}_{\mathrm{Reg}}(C)$, which takes as input a constraint C and returns the corresponding RE: E.g., $\mathscr{E}_{\mathrm{Reg}}(Response(a, b))$=`[^a]*(a.*b)*[^a]*`. If we consider the operations of conjunction between DECLARE constraints ($\wedge$) and intersection between REs (&), $\mathscr{E}_{\mathrm{Reg}}$ is a monoid homomorphism w.r.t. $\wedge$ and &. In other words, given two constraints C and C', $\mathscr{E}_{\mathrm{Reg}}(C \wedge C') = \mathscr{E}_{\mathrm{Reg}}(C)$ & $\mathscr{E}_{\mathrm{Reg}}(C')$, preserving closure, associativity and the identity element (resp., $\top$ and `.*`).

As mentioned in Section 3.4, an RE can always be associated to a deterministic labelled FSA, which accepts all and only those finite strings that match the RE. We name as $\mathscr{A}$ the operation leading from an RE to an FSA, thus we have that a DECLARE constraint can be associated with its corresponding FSA, $A^C = \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C))$. Henceforth, we also call A^C the C-automaton. Under this interpretation, a constraint C is evaluated to true over a trace if and only if its events can be replayed as a finite path on the C-automaton that terminates in an accepting state, i.e., when its corresponding string is accepted by the C-automaton.

Considering an example declarative process model $\mathsf{M} = \langle \mathsf{A}, \{Init, AlternateResponse, ChainPrecedence, \ldots\}, \Gamma \rangle$ with $\mathsf{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$

and $\Gamma = \{Init(\mathsf{a}), AlternateResponse(\mathsf{a}, \mathsf{b}), ChainPrecedence(\mathsf{b}, \mathsf{c})\}$, the C-automata of the set of constraints Γ are drawn in Figure 3: $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(Init(\mathsf{a})))$, $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(AlternateResponse(\mathsf{a}, \mathsf{b})))$, and $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(ChainPrecedence(\mathsf{b}, \mathsf{c})))$ are depicted in Figure 3(a), Figure 3(b), and Figure 3(c), respectively. Figure 4 shows the product automata that are derived from the intersection of such constraints: $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(Init(\mathsf{a}))) \times \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(AlternateResponse(\mathsf{a}, \mathsf{b})))$ is illustrated in Figure 4(a), and $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(Init(\mathsf{a}))) \times \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(AlternateResponse(\mathsf{a}, \mathsf{b}))) \times \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(ChainPrecedence(\mathsf{b}, \mathsf{c})))$ is illustrated in Figure 4(b).

We remark that by applying $\mathscr{A}$ to the RE of a conjunction of constraints, we obtain an FSA that corresponds to the product $\times$ of the FSAs for the individual constraints [36]: $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C \wedge C')) = \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C)) \times \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C'))$. Also, we recall that the identity element for FSAs is a single-state automaton whose unique state is both initial and accepting, and has a self-loop for each character in the considered alphabet.

Given a model $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$, we can therefore implicitly describe the set of traces that comply with $\mathcal{M}$ as the language accepted by the product of all C-automata (one for every $C \in \Gamma$). In the light of this discussion, our approach searches a solution to the problem of finding a non-redundant consistent DECLARE model within the **automata-product monoid**, i.e., the associative algebraic structure with identity element (the universe-set of FSAs) and product operation $\times$. For the automata-product monoid, the property of commutativity holds.

5.2. Sorting Constraints

The objective of the algorithm is to visit the constraints in the declarative process model only once. At every visit, the analysed constraint is checked whether it is conflicting or redundant. In the first case, it is relaxed (replaced by the subsuming constraint) and checked again. If the constraint is not subsumed by any another, it is removed from the set of constraints. In case of redundancy, the constraint under analysis is removed from the set of discovered constraints. The order in which the constraints are checked is thus of utmost importance, as it determines their priority. The priority, in turn, implicitly defines the "survival expectation" of a constraint, as constraints that come later in the list are more likely to be pruned if they are either redundant or conflicting.

We identify four notions of ordering relations for declarative process models. Three of them are suitable for all models, i.e., *(i)* order on the degree

24

of activation linkage $\leqslant_{\leadsto}$, *(ii)* partial order on the type $\leqslant_{\mathrm{T}}$, and *(iii)* partial order on the subsumption $\leqslant_{\sqsupseteq}$; the last one is specific for discovered models, i.e., *(iv)* order on support, confidence, and interest factor $\leqslant_{\sigma\kappa\iota}$.

The first ordering relation is based on the notion of degree of activation linkage. Given a constraint C in the template instantiation relation Γ of a model $\mathcal{M}$ and its activation $C|_{\bullet}$, it counts the number of tasks that play the role of target in constraints that share the same activation of C. Formally,

$$\leadsto(C,\Gamma) = |\{a \in \mathcal{A} : \exists C' \in \Gamma \text{ s.t. } a = C'|_{\Rightarrow} \text{ and } C|_{\bullet} = C'|_{\bullet}\}|. \qquad (8)$$

Recalling that we consider for unary constraints the activation to coincide with the target, for the example model $\mathsf{M} = \langle \mathsf{A}, \{Init, AlternateResponse, ChainPrecedence, \ldots\}, \Gamma\rangle$, where $\mathsf{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and $\Gamma = \{Init(\mathsf{a}), AlternateResponse(\mathsf{a},\mathsf{b}), ChainPrecedence(\mathsf{b},\mathsf{c})\}$, we have that *(i)* $\leadsto(Init(\mathsf{a}),\Gamma) = 2$, *(ii)* $\leadsto(AlternateResponse(\mathsf{a},\mathsf{b}),\Gamma) = 2$, *(iii)* $\leadsto(ChainPrecedence(\mathsf{b},\mathsf{c}),\Gamma) = 1$. The (total) order on the degree of activation linkage is thus defined as follows:

$$C \leqslant_{\leadsto} C' \iff \leadsto(C,\Gamma) \leqslant \leadsto(C',\Gamma), \text{ with } C, C' \in \Gamma. \qquad (9)$$

This relation is meant to sort constraints by the number of tasks that are subject to conditions over the execution of their activation.

The partial order on the type of constraints is driven by the expertise acquired in the last years in the context of DECLARE discovery [30, 19]. In particular, we tend to preserve those constraints that have the potential of inducing the removal of a massive amount of other constraints due to redundancy. As an example, consider the case of the *Init* template: Given $a \in \mathcal{A}$, if $Init(a)$ holds true, then also the relation constraint $Precedence(a, b)$ is guaranteed to hold true for every $b \in \mathcal{A}\setminus\{a\}$. This means that, in the best case, $|\mathcal{A}|-1$ constraints will be removed because they are all redundant with $Init(a)$. Similarly, consider the positive relation constraint $ChainResponse(a, b)$: It implies $NotChainSuccession(a, c)$ for every $c \in \mathcal{A}\setminus\{a, b\}$. Thus, $ChainResponse(a, b)$ has the potential of triggering the removal of $|\mathcal{A}|-2$ negative constraints due to redundancy. Therefore, the ordering by type sorts constraints according to the following ranking from the highest to the lowest:

5. position constraints,

4. cardinality constraints,

3. coupling constraints,

2. forward- and backward-unidirectional relation constraints,

1. negative constraints.

We define the partial order on the subsumption as follows:

$$C \leqslant_{\sqsupseteq} C' \iff C \sqsupseteq C' \tag{10}$$

Those constraints that have the highest likelihood to induce other constraints are ranked the highest by both the last two orderings. Therefore, their application seems to be suitable to prune out the highest number of constraints, especially during the redundancy check. Therefore, we introduce the hybrid ordering relation $\leqslant_{T\sqsupseteq}$, defined as follows:

$$C \leqslant_{T\sqsupseteq} C' \equiv (C \leqslant_T C' \wedge \neg (C' \leqslant_T C)) \vee C \leqslant_{\sqsupseteq} C'. \tag{11}$$

In essence, it compares the type of constraints C and C'. If they are the same, i.e., $C \leqslant_T C' \wedge \neg (C' \leqslant_T C))$ holds true, then the comparison is made on the basis of $\leqslant_{\sqsupseteq}$.

Finally, for discovered declarative process models, functions σ, κ and ι are defined for constraints. Therefore, the last ordering relation based on these functions can be applied:

$$C \leqslant_{\sigma\kappa\iota} C' \iff \sigma(C, L) \leqslant \sigma(C', L) \vee \kappa(C, L) \leqslant \kappa(C', L) \vee \iota(C, L) \leqslant \iota(C', L). \tag{12}$$

Such an ordering is meant to give priority to those constraints that are violated the least within the event log or whose constrained activities occur most frequently. The idea is to remove those constraints that are redundant or conflicting starting from those that are less fitting with the event log.

The aforementioned ordering relations are not strict, because they are not asymmetric: For instance, in the example given for the order on the degree of activation linkage, both $Init(\mathsf{a}) \leqslant_{\rightsquigarrow} AlternateResponse(\mathsf{a}, \mathsf{b})$ and $AlternateResponse(\mathsf{a}, \mathsf{b}) \leqslant_{\rightsquigarrow} Init(\mathsf{a})$ hold true. Since the number of constraints in a model is finite, we can assume the existence of a strict total order over constraints $\leqslant_{\#}$ s.t. only one of the following three statements holds for every pair of constraints C, C': Either *(i)* $C \leqslant_{\#} C'$, or *(ii)* $C' \leqslant_{\#} C$, or *(iii)* $C = C'$. Relation $\leqslant_{\#}$ can be based, e.g., on a perfect hash relation, or on an enumeration-based ordering of constraints. This notion allows us to
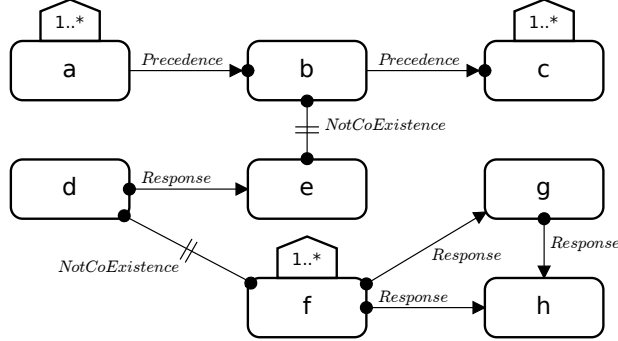
Figure 5: An example of a DECLARE model.

postulate the creation of a strict total order relation $\leqslant$ based on a sequential application of any combination $\langle \leqslant_1, \ldots, \leqslant_n \rangle$ of the aforementioned ordering relations $\leqslant_\leadsto$, $\leqslant_{\top\sqsupseteq}$, and $\leqslant_{\sigma\kappa\iota}$. This is inductively defined as follows:

$$C \underset{\langle \leqslant_1, \ldots, \leqslant_n \rangle}{\leqslant} C' \equiv \bigvee_{i \in [1, n-1]} (C \leqslant_i C' \wedge C' \leqslant_i C) \ \vee \ C \leqslant_{i+1} C' \ \vee \ (C \leqslant_\# C').$$

(13)

This relation orders constraints by applying the first ordering relation $\leqslant_1$ to the pair of constraints (C, C'). If they are such that $C \leqslant_1 C'$ and $C' \leqslant_1 C$, then the comparison by means of $\leqslant_2$ is applied, and so forth till the n-th stage is reached. Then, $\leqslant_\#$ is applied. By definition, $\leqslant_\#$ is a strict total order, therefore it holds that $C \leqslant_\# C'$ and $C' \leqslant_\# C$ if and only if $C = C'$.

We introduce an algorithm henceforth referred to as $sortBy\left(\Gamma, \langle \leqslant_1, \ldots, \leqslant_n \rangle, s^{\rightleftarrows}\right)$. Its input consists of *(i)* a set of constraints Γ, *(ii)* a set of ordering relation symbols $\langle \leqslant_1, \ldots, \leqslant_n \rangle$ over constraints, and *(iii)* a constant $s^{\rightleftarrows} \in \{\text{ASC}, \text{DESC}\}$, specifying whether the sort has to be performed in an ascending (ASC) or descending (DESC) order. It returns a list of constraints ordered on the basis of the strict total order relation $\underset{\langle \leqslant_1, \ldots, \leqslant_n \rangle}{\leqslant}$.

Figure 5 depicts the following example DECLARE model:

$$M = \langle A = \{a, b, c, d, e, f, g, h\},$$

$$\mathfrak{C} = \{Participation, Precedence, NotCoExistence, Response, \ldots\}$$

$$\Gamma = \{Participation(a), Precedence(a, b),$$

$$Precedence(b, c), NotCoExistence(b, e),$$

$$Participation(c),$$

$$Response(d, e),$$

$$Response(f, g), Participation(f), NotCoExistence(f, d), Response(f, h),$$

$$Response(g, h)\}\rangle.$$

The application of *sortBy* on Γ with the ordering relation symbols $\leqslant_{\rightsquigarrow}$ and $\leqslant_{T\sqsupseteq}$ returns the following list:

$$sortBy(\Gamma, \{\leqslant_{\rightsquigarrow}, \leqslant_{T\sqsupseteq}\}, \text{DESC}) = \langle Participation(f),$$

$$Response(f, g),$$

$$Response(f, h),$$

$$NotCoExistence(f, d),$$

$$Precedence(a, b),$$

$$NotCoExistence(b, e),$$

$$Participation(c),$$

$$Precedence(b, c),$$

$$Response(d, e),$$

$$Participation(a),$$

$$Response(g, h)\rangle.$$

The order of constraints deeply affects the way in which the proposed algorithm verifies whether they are contradicting or redundant. The algorithm indeed iterates over the list and checks at every step the current constraint against the ones that have already been processed. Reading the constraints as returned by the $\{\leqslant_{T\sqsupseteq}, \leqslant_{\rightsquigarrow}\}$ descending sorting, it can be verified that $Response(d, e)$ is classified as redundant as well as $Participation(a)$. The reason why $Response(d, e)$ is recognised as redundant resides in the fact that it imposes that e occurs after d. However, the already visited constraint $Participation(f)$ and $NotCoExistence(f, d)$ respectively

state that f always occur and that its occurrence implies d to not occur at all. Therefore, specifying conditions based on the occurrence of task (d) is redundant because d cannot occur. For what $Participation(\mathsf{a})$ is concerned, it is classified as redundant because of the already visited constraints $Precedence(\mathsf{a},\mathsf{b})$, $Participation(\mathsf{c})$, and $Precedence(\mathsf{b},\mathsf{c})$. They specify that c must always occur ($Participation(\mathsf{c})$), and that if c occurs, then b must precede it ($Precedence(\mathsf{b},\mathsf{c})$). Hence, b always occurs too. $Precedence(\mathsf{a},\mathsf{b})$ specifies that b cannot occur if it is not preceded by a. Thus, a must occur as well and $Participation(\mathsf{a})$ can be classified as redundant.

Notice that $Response(\mathsf{f},\mathsf{h})$ is redundant too. In fact, $Response(\mathsf{f},\mathsf{g})$ specifies that if f occurs, then g must occur too. $Response(\mathsf{g},\mathsf{h})$ imposes that, after g, h must occur. As a consequence, after f, h must also occur. However, $Response(\mathsf{g},\mathsf{h})$ is checked only as the very last constraint in the list, hence after $Response(\mathsf{f},\mathsf{h})$. Therefore, $Response(\mathsf{f},\mathsf{h})$ cannot be recognised as redundant. The redundancy would be detected if a second iteration was conducted over the list, by considering whether the current constraint is already implied by all the others. This is the reason why our approach provides for such a second check, which comes at the price of a slower computation though. Notice that we execute the second iteration from the last element to the first one so that constraints with lower priority are processed (and in case eliminated) first.

Finally, we remark here that there is no risk of overlooking contradicting constraints. This can be intuitively explained by the fact that a contradicting constraint always leads to an empty model, regardless of whether it is evaluated as first or last. The different order can only affect *which* constraint among the ones in conflict is checked last and hence classified as contradicting.

5.3. The Algorithm

Algorithm 1 outlines the pseudocode of our technique. Its input consists of: *(i)* A DECLARE model $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$ discovered from an event log $L \in \mathbb{M}(\mathcal{A}^*)$, bearing a set of constraints Γ defined over log alphabet $\mathcal{A}$ and repertoire $\mathfrak{C}$, *(ii)* a list of ordering relation symbols $\langle \leqslant_1, \ldots, \leqslant_n \rangle$, and *(iii)* a boolean flag r^{II} specifying whether a second redundancy check has to be performed or not. For every $C \in \Gamma$, we assume that its support, confidence, and interest factor are given too, which is the usual condition when $\mathcal{M}$ is the output of mining algorithms such as Declare Maps Miner

Algorithm 1: Algorithm $makeConsistent\left(\mathcal{M}, \langle \leqslant_1, \ldots, \leqslant_n \rangle, r^{\mathrm{II}}\right)$, returning the suboptimal solution to the problem of finding a minimal set of non-conflicting constraints in a discovered DECLARE model. Its input consists of a declarative process model $\mathcal{M} = \langle \mathcal{A}, \mathfrak{C}, \Gamma \rangle$, a list of ordering relation symbols $\langle \leqslant_1, \ldots, \leqslant_n \rangle$, and a boolean flag r^{II}, enabling a second check for redundancies.

Input: A DECLARE model $\mathcal{M} = \{\mathcal{A}, \mathfrak{C}, \Gamma\}$, defined over $\mathcal{A}$. $\mathcal{M}$ is a set of constraints for which support, confidence and interest factor are given
Output: Set of non-conflicting constraint Γ^{R}

```
/* Initialisation phase                                                    */
```
1   $\Gamma' \leftarrow removeSubsumptionHierarchyRedundancies(\Gamma)$
2   $\Gamma^{\mathrm{S}} \leftarrow \{C \in \Gamma' : \sigma = 1.0\}$ `// Non-conflicting constraints`
3   $\Gamma^{\mathrm{U}} \leftarrow \Gamma' \setminus \Gamma^{\mathrm{S}}$ `// Potentially conflicting constraints`
4   $A \leftarrow \langle \mathcal{A}, \{s_0\}, s_0, \{\bigcup_{\sigma \in \mathcal{A}} \langle s_0, \sigma, s_0 \rangle, \{s_0\}\} \rangle$ `// Automaton accepting any sequence of tasks`
5   $\Gamma^{\mathrm{R}} \leftarrow \emptyset$ `// Set of returned constraints`
6   $\Gamma^{\mathrm{V}} \leftarrow \emptyset$ `// Set of checked constraints`

```
/* Prune redundant constraints from the set of non-conflicting ones        */
```
7   $\Gamma^{\mathrm{S}}_{\mathrm{list}} \leftarrow sortBy(\Gamma^{\mathrm{S}}, \langle \leqslant_1, \ldots, \leqslant_n \rangle, \mathrm{DESC})$ `// Sort constraints in $\Gamma^{\mathrm{S}}$ in descending order`
8   **foreach** $C_i^{\Gamma^{\mathrm{S}}} \in \Gamma^{\mathrm{S}}_{\mathrm{list}}$,   with $i \in [1, |\Gamma^{\mathrm{S}}_{\mathrm{list}}|]$ **do**
9      $\Gamma^{\mathrm{V}} \leftarrow \Gamma^{\mathrm{V}} \bigcup \left\{ C_i^{\Gamma^{\mathrm{S}}} \right\}$ `// Record that $C_i^{\Gamma^{\mathrm{S}}}$ has been checked`
10      $A^{C_i^{\Gamma^{\mathrm{S}}}} \leftarrow \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}\left(C_i^{\Gamma^{\mathrm{S}}} \right) \right)$ `// Build the constraint-automaton of $C_i^{\Gamma^{\mathrm{S}}}$`
11   **if** $\mathscr{L}(A) \supset \mathscr{L}\left(A^{C_i^{\Gamma^{\mathrm{S}}}} \right)$ **then**   `// If $C_i^{\Gamma^{\mathrm{S}}}$ is not redundant`
12          $A \leftarrow A \times A^{C_i^{\Gamma^{\mathrm{S}}}}$ `// Merge the $C_i^{\Gamma^{\mathrm{S}}}$-automaton with the main FSA`
13          $\Gamma^{\mathrm{R}} \leftarrow \Gamma^{\mathrm{R}} \bigcup \left\{ C_i^{\Gamma^{\mathrm{S}}} \right\}$ `// Include $C_i^{\Gamma^{\mathrm{S}}}$ in the set of returned constraints`

```
/* Pruning of conflicting constraints                                      */
```
14   $\Gamma^{\mathrm{U}}_{\mathrm{list}} \leftarrow sortBy(\Gamma^{\mathrm{U}}, \langle \leqslant_1, \ldots, \leqslant_n \rangle, \mathrm{DESC})$ `// Sort constraints in $\Gamma^{\mathrm{U}}$ in descending order`
15   **foreach** $C_i^{\Gamma^{\mathrm{U}}} \in \Gamma^{\mathrm{U}}_{\mathrm{list}}$,   with $i \in [1, |\Gamma^{\mathrm{U}}_{\mathrm{list}}|]$ **do**
16      $resolveConflictAndRedundancy\left(A, \Gamma^{\mathrm{R}}, C_i^{\Gamma^{\mathrm{U}}}, \Gamma^{\mathrm{V}} \right)$

```
/* Second redundancy check                                                 */
```
17   **if** $r^{\mathrm{II}} = \top$ **then**
18      $resolveRedundancies^{\mathrm{II}}\left(\Gamma^{\mathrm{R}}, \langle \leqslant_1, \ldots, \leqslant_n \rangle \right)$

19   **return** $removeSubsumptionHierarchyRedundancies(\Gamma^{\mathrm{R}})$

---

or MINERful. Table 2(a) shows an example of Γ, i.e., Γ, defined on the log alphabet $\{a, b, c, d\}$. We also assume that the same metrics are defined for those constraints that are not in $\mathcal{M}$, yet are either their subsuming, negated, forward or backward versions. For the sake of readability, these additional constraints are not reported in Table 2. Table 2(b) shows the output that corresponds to the post-processing of Table 2(a), provided that the ordering relation symbols provided are $\langle \leqslant_{\sigma\kappa\iota}, \leqslant_{T\sqsupseteq} \rangle$ and r^{II} is false. Constraints that are considered as redundant are coloured in grey. Struck-out constraints are those that are in conflict with the others and thus dropped from the returned set.

Given $\mathcal{M}$ and its constraints set Γ, the first operation *removeSubsumptionHierarchyRedundancies* prunes out redundant constraints from Γ based on the subsumption hierarchy. The procedure removes the subsuming constraints if their support is less than or equal to the subsumed ones. Forward and backward constraints are also eliminated if the corresponding coupling constraint has an equivalent support. The result is stored in Γ'. The details of this operation have already been described in [28]. The usefulness of this procedure resides in the fact that it reduces the number of candidate constraints to be considered, thus reducing the number of iterations performed by the algorithm. In Table 2(b), this operation is responsible for the elimination of *Participation*(a), due to the fact that *Init*(a) is known to hold true.

Thereafter, we partition Γ' into two subsets, i.e.: *(i)* Γ^{S} consisting of those constraints that are verified over the entire event log (i.e., having a support of 1.0), and *(ii)* Γ^{U} containing the remaining constraints. The reason for doing this is that the former is guaranteed to have no conflict: Given the fact that constraints are discovered using the alphabet of the event log, those that have a support of 1.0 can be joined, giving rise to a *consistent* constraint model.

Even though constraints in Γ^{S} are guaranteed to be conflict-free, they could still contain redundancies. Therefore, the following part of the algorithm is dedicated to the elimination of redundant constraints from this set. To check redundancies, we employ the characterisation of constraints in terms of FSAs. Instead, constraints in Γ^{U} may contain both redundancies and inconsistencies. Table 2(b) presents the partition of $\mathcal{M}$ into Γ^{S} and Γ^{U}.

First, we initialise an FSA A to be the identity element w.r.t. the automata product. In other words, A is initialised to accept any sequence of events that map to a task in the log alphabet. This automaton incrementally

Constraint	σ	κ	ι
$Init(\mathtt{a})$	1.0	1.0	1.0
$Participation(\mathtt{a})$	1.0	1.0	1.0
$CoExistence(\mathtt{a},\mathtt{d})$	1.0	1.0	1.0
$End(\mathtt{d})$	1.0	1.0	1.0
$NotChainSuccession(\mathtt{b},\mathtt{d})$	1.0	0.9	0.8
$NotChainSuccession(\mathtt{a},\mathtt{d})$	0.75	0.5	0.5
$ChainResponse(\mathtt{b},\mathtt{c})$	1.0	0.9	0.8
$NotChainSuccession(\mathtt{a},\mathtt{b})$	0.9	0.7	0.6
$NotChainSuccession(\mathtt{a},\mathtt{c})$	0.8	0.7	0.6
$ChainResponse(\mathtt{b},\mathtt{a})$	0.75	0.9	0.9

(a) Input

	i	Constraint	σ	κ	ι
$\Gamma^{\mathrm{S}}_{\mathrm{list}}$	1	$Init(\mathtt{a})$	1.0	1.0	1.0
	2	$End(\mathtt{d})$	1.0	1.0	1.0
	3	$CoExistence(\mathtt{a},\mathtt{d})$	1.0	1.0	1.0
	4	$ChainResponse(\mathtt{b},\mathtt{c})$	1.0	0.9	0.8
	5	$NotChainSuccession(\mathtt{b},\mathtt{d})$	1.0	0.9	0.8
$\Gamma^{\mathrm{U}}_{\mathrm{list}}$	1	$NotChainSuccession(\mathtt{a},\mathtt{b})$	0.9	0.7	0.6
	2	$NotChainSuccession(\mathtt{a},\mathtt{c})$	0.8	0.7	0.6
	4	$AlternateResponse(\mathtt{b},\mathtt{a})$	0.75	0.9	0.9
	3	$NotChainSuccession(\mathtt{a},\mathtt{d})$	0.75	0.5	0.5

(b) Processed output

Table 2: Example of input constraint set processing.

incorporates the constraints of the input model based on their priority. To set up redundancy elimination in Γ^{S} as well as redundancy and inconsistency elimination in Γ^{U}, we then order their constitutive constraints according to the criteria specified by the user, $\langle \leqslant_1, \ldots, \leqslant_n \rangle$. The ranking determines the priority with which constraints are analysed.

After the sorting, constraints are stepwise considered for inclusion in the refined model by iterating over the corresponding ranked lists. Constraints in Γ^{S}, i.e., $C_i^{\Gamma^{\mathrm{S}}} \in \Gamma^{\mathrm{S}}_{\mathrm{list}}$, are only checked for redundancy, whereas constraints in Γ^{U}, $C_i^{\Gamma^{\mathrm{U}}} \in \Gamma^{\mathrm{U}}_{\mathrm{list}}$, are checked for both redundancy and consistency. For every constraint $C_i^{\Gamma^{\mathrm{S}}} \in \Gamma^{\mathrm{S}}_{\mathrm{list}}$, redundancy is checked by leveraging language inclusion. In particular, this is done by computing the FSA $A^{C_i^{\Gamma^{\mathrm{S}}}}$ for $C_i^{\Gamma^{\mathrm{S}}}$ and then checking whether its generated language $\mathscr{L}\left(A^{C_i^{\Gamma^{\mathrm{S}}}}\right)$ is included inside $\mathscr{L}(A)$, which considers the contribution of all constraints processed so far. If this is the case, then the constraint is dropped. Otherwise, A is extended with the contribution of this new constraint (by computing the product $A \times A^{C_i^{\Gamma^{\mathrm{S}}}}$) and $C_i^{\Gamma^{\mathrm{S}}}$ is added to the set Γ^{R} of constraints to be returned. In the example of Table 2(b), $CoExistence(\mathtt{a},\mathtt{d})$ is analysed after the existence constraints $Init(\mathtt{a})$ and $End(\mathtt{d})$ based on the preliminary sorting operation. It thus turns out to be redundant, because $Init(\mathtt{a})$ and $End(\mathtt{d})$ already specify that both $\mathtt{a}$ and $\mathtt{d}$ will occur in every trace. Therefore, they will necessarily always co-occur.

Redundancy and consistency checks of constraints $C_i^{\Gamma^{\mathrm{U}}} \in \Gamma^{\mathrm{U}}_{\mathrm{list}}$ is performed by the *resolveConflictAndRedundancy* procedure (Algorithm 2). The procedure checks the consistency of those constraints that are not redun-

---

Algorithm 2: Algorithm $resolveConflictAndRedundancy\left(A,\Gamma^{\mathrm{R}},C,\Gamma^{\mathrm{V}}\right)$, adding a constraint C to the set of constraint Γ^{R}, if it has not already been checked (and thus included in Γ^{V}), and is neither conflicting nor redundant with the already added constraints.

Input: An FSA A, a set of non-conflicting constraints Γ^{R}, a constraint C, and a list of already checked constraints Γ^{V}

1 **if** $C \notin \Gamma^{\mathrm{V}}$ **then**   // If C was not already checked
2     $\Gamma^{\mathrm{V}} \leftarrow \Gamma^{\mathrm{V}} \bigcup \{C\}$ // Record that C has been checked
3     $A^{C} \leftarrow \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}(C)\right)$ // Build the C-automaton
4   **if** $\mathscr{L}(A) \supset \mathscr{L}\left(A^{C}\right)$ **then**   // If C is not redundant
5   **if** $\mathscr{L}\left(A \times A^{C}\right) \neq \emptyset$ **then**   // If C is not conflicting
6           $A \leftarrow A \times A^{C}$ // Merge the C-automaton with the main FSA
7           $\Gamma^{\mathrm{R}} \leftarrow \Gamma^{\mathrm{R}} \bigcup \{C\}$ // Include C in the set of returned constraints

8   **else**   // Otherwise, resolve the conflict
9   **if** $\mathscr{R}(C) \neq \top$ **then**   // If a relaxation of C, i.e., $\mathscr{R}(C)$, exists
10           $resolveConflictAndRedundancy\left(A,\Gamma^{\mathrm{R}},\mathscr{R}(C),\Gamma^{\mathrm{V}}\right)$

11   **if** C *is a coupling constraint* **then**
12           $resolveConflictAndRedundancy\left(A,\Gamma^{\mathrm{R}},fw(C),\Gamma^{\mathrm{V}}\right)$
13           $resolveConflictAndRedundancy\left(A,\Gamma^{\mathrm{R}},bw(C),\Gamma^{\mathrm{V}}\right)$

---

dant. The redundancy is, again, checked based on the language inclusion of the language generated by the currently analysed constraint $\mathscr{L}\left(A^{C_i^{\Gamma^{\mathrm{U}}}}\right)$ in $\mathscr{L}(A)$, where A is the automaton that accumulates the contribution of all constraints that have been kept so far. The consistency is checked through a language emptiness test performed over the intersection of $\mathscr{L}\left(A^{C_i^{\Gamma^{\mathrm{U}}}}\right)$ and $\mathscr{L}(A)$. This is done by checking that $\mathscr{L}\left(A \times A^{C_i^{\Gamma^{\mathrm{U}}}}\right) \neq \emptyset$. In case a conflict is detected, we do not immediately drop the conflicting constraint, but we try, instead, to find a more relaxed constraint that retains its intended semantics as much as possible, but does not incur in a conflict. To do so, we use the constraint subsumption hierarchy. In particular, we use the relaxation operator to retrieve the parent constraint of the conflicting one, and we recursively invoke the *resolveConflictAndRedundancy* procedure over the parent. The recursion terminates when the first non-conflicting ancestor of the conflicting constraint is found or when the top of the hierarchy is reached. The two cases are resp. covered in the example of Table 2(b) by *ChainResponse*(b, a), replaced by *AlternateResponse*(b, a), and by *NotChainSuccession*(a, d), which is removed because a non-conflicting

ancestor does not exists. Note that $NotChainSuccession(\mathtt{a},\mathtt{d})$ is to be eliminated because of the interplay of the other two $NotChainSuccession$ constraints, $Init(\mathtt{a})$ and $End(\mathtt{d})$. $ChainResponse(\mathtt{b},\mathtt{a})$ is in conflict with $ChainResponse(\mathtt{b},\mathtt{c})$.

If the constraint under analysis is a coupling constraint, then we know that it is constituted by the conjunction of a corresponding pair of forward and backward constraints. In this situation, it could be the case that all the relaxations of the coupling constraint along the subsumption hierarchy continue to be conflicting, but the conflict would be removed by just considering either its forward or backward component (or a relaxation thereof). Consequently, we also recursively invoke the *resolveConflictAndRedundancy* procedure on these two components.

Algorithm 3: Algorithm $resolveRedundancies^{\mathrm{II}}\left(\Gamma^{\mathrm{R}}, \langle \leqslant_1, \ldots, \leqslant_n \rangle\right)$, removing from Γ^{R} those constraints that are redundant. The check is applied once for every constraint in Γ^{R}, in ascending order w.r.t. the criteria specified by $\langle \leqslant_1, \ldots, \leqslant_n \rangle$.

Input: A set of non-conflicting constraints Γ^{R} and a list of ordering relation symbols $\langle \leqslant_1, \ldots, \leqslant_n \rangle$

/* Sort constraints in Γ^{R} in ascending order   */

1    $\Gamma^{\mathrm{R}}_{\mathrm{list}} \leftarrow sortBy\left(\Gamma^{\mathrm{R}}, \langle \leqslant_1, \ldots, \leqslant_n \rangle, \mathrm{ASC}\right)$

/* Build the product-automaton of constraints in Γ^{R} */

2    $A^{\mathrm{R}} \leftarrow \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}(C'_1)\right) \times \ldots \times \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}(C'_l)\right)$ with $C'_1, \ldots C'_l \in \Gamma^{\mathrm{R}}$ and $l = \left|\Gamma^{\mathrm{R}}\right|$

/* Resolve redundancies   */

3   **foreach** $C \in \Gamma^{\mathrm{R}}_{\mathrm{list}}$ **do**

4      $\Gamma^{\mathrm{R}}_{\not\subseteq} \leftarrow \bigcup\limits_{C' \in \Gamma^{\mathrm{R}}_{\mathrm{list}} \setminus \{C\}} \{C'\}$ // All constraints in Γ^{R} except C

      /* Build the product-automaton of all constraints in Γ^{R} except C */

5      $A^{\mathrm{R}}_{\not\subseteq} \leftarrow \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}(C''_1)\right) \times \ldots \times \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}(C''_m)\right)$ with $C''_1, \ldots C''_m \in \Gamma^{\mathrm{R}}_{\not\subseteq}$ and $m = \left|\Gamma^{\mathrm{R}}_{\not\subseteq}\right|$

6   **if** $\mathscr{L}\left(A^{\mathrm{R}}_{\not\subseteq}\right) \subseteq \mathscr{L}(A^{\mathrm{R}})$ **then**   // If C is redundant

7         $\Gamma^{\mathrm{R}} \leftarrow \Gamma^{\mathrm{R}} \setminus \{C\}$ // Remove C from the set of returned constraints

---

To limit the issue of missing some redundancies, due to the single iterative check over the elements in the constraint lists, a second check can be performed. The technique is slightly different from the one applied before and requires more computational time. For this reason, *(i)* it is performed after the first iteration has taken place, so as to diminish the amount of constraints to be verified, and *(ii)* it is an optional functionality, activated by the user-specified flag r^{II}. Its pseudocode is listed in Algorithm 3. It takes as input a set of non-conflicting constraints Γ^{R} and a list of order relation

symbols $\langle \leqslant_1, \ldots, \leqslant_n \rangle$. First, the product-automaton A^{R} is built that consists of the product of all constraint-automata of elements in Γ^{R}. Thereafter, constraints in Γ^{R} are sorted according to $\langle \leqslant_1, \ldots, \leqslant_n \rangle$ in ascending order, i.e., the reverse order w.r.t. the first pass (this is because constraints with a lower priority are processed first and eliminated if redundant). The ordered list that comes out of the sorting is $\Gamma^{\mathrm{R}}_{\mathrm{list}}$. For each constraint C in $\Gamma^{\mathrm{R}}_{\mathrm{list}}$, a new product-automaton is built that considers all constraints in Γ^{R} except C. It is hereinafter indicated as $\Gamma^{\mathrm{R}}_{\not{C}}$. The redundancy check is thus performed: If the language accepted by $\Gamma^{\mathrm{R}}_{\not{C}}$ is a subset of the language accepted by Γ^{R}, then the declarative process model excluding C is as restrictive as the declarative process model including C. This means that C can be classified as redundant.

The algorithm proceeds iteratively for all remaining constraints in $\Gamma^{\mathrm{R}}_{\mathrm{list}}$. Although every element in the list is visited once, the overall procedure is expensive in terms of computation time because a new product-automaton must be built at every step by the cross-product of $l-1$ constraint-automata, where l is the cardinality of the set of input constraints Γ^{R}. The procedure cannot take advantage of the associativity of the cross-product operation, because temporary automata are built at every step without storing the intermediate result of $l-2$ cross-products. However, the additional computational effort is compensated by a higher accuracy in the redundancy-check: In the example of Section 5.2, the redundant constraint $Response(\mathsf{f},\mathsf{h})$ would be detected by the second-pass algorithm, whilst it was not captured in the first check.

Finally, a last complete pass over constraints in Γ^{R} is done, to check again whether there are subsumption-hierarchy redundancies. If so, Γ^{R} is pruned accordingly.

Complexity of the Algorithm. We close this section by elaborating on the complexity of the algorithm, considering as input the *number* of constraints contained in the discovered model. To better highlight the different sources of complexity, we consider the overall complexity as well as the complexity obtained from the crude algorithm, without considering the contribution of the automata-manipulating operations. This is particularly important because, even though in the worst case the automata-manipulating operations are exponential in the number of constraints, in practice, they have a nonmonotonic behavior. Consider, for example, the cross-product operation between an automaton A and the automaton $A^C = \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C))$ of constraint C: It

is not guaranteed that $A \times A^C$ has a number of states bigger than that of A. This is witnessed, e.g., by the automaton in Figure 6: The size of the automaton of Figure 4(b), generated by $Init(\mathsf{a})$, $AlternateResponse(\mathsf{a},\mathsf{b})$, and $ChainPrecedence(\mathsf{b},\mathsf{c})$, is bigger than the size of its cross-product with the automaton of $AtMostOne(\mathsf{c})$.

Theorem 5.1. *Given a* Declare *model* $\mathcal{M}$ *containing* n *constraints, a list* $\langle \leqslant_1, \ldots, \leqslant_n \rangle$ *of ordering relation symbols, and a boolean flag* r^{II}, *algorithm* $makeConsistent\left(\mathcal{M}, \langle \leqslant_1, \ldots, \leqslant_n \rangle, r^{II}\right)$ *runs in time*
- $O(2^n)$, *if the automata-manipulating operations are considered part of the algorithm;*
- $O(n^2)$, *if* r^{II} *is true (i.e., the algorithm includes redundancy double check) and the automata-manipulating operations are considered not part of the algorithm;*
- $O(n \cdot \log(n))$, *if* r^{II} *is false (i.e., the algorithm skips redundancy double check) and the automata-manipulating operations are considered not part of the algorithm.*

Proof. As shown in [28], operation *removeSubsumptionHierarchyRedundancies* (lines 1 and 18 in Algorithm 1) requires a check based on a depth-first visit on a subsumption hierarchy's direct acyclic graph (see Figure 1) for every constraint in the input model. For the coupling constraints, also their related forward- and backward-unidirectional relation constraints are considered. The hierarchy structure is however fixed and the number of steps for the visit are thus limited. In particular, the worst case is represented by *ChainSuccession*(a, b), as it can be seen in 1. It requires at most 6 comparisons: 4 for the subsuming constraints, i.e., *AlternateSuccession*(a, b), *Succession*(a, b), *CoExistence*(a, b), *RespondedExistence*(a, b), and 2 for the forward- and backward-unidirectional relation constraint, resp. *ChainResponse*(a, b) and *ChainPrecedence*(a, b). Both invocations thus cost $O(n)$. By construction, operation *removeSubsumptionHierarchyRedundancies* returns a set of constraints Γ' such that $|\Gamma'| \leqslant n$.
Lines 7 and 14 of Algorithm 1 both apply a sorting algorithm to sets of constraints Γ^S and Γ^U, respectively. Notice that $|\Gamma^S| \leqslant n$ and $|\Gamma^U| \leqslant n$. An efficient algorithm such as merge-sort consequently requires $O(n \cdot \log(n))$ for this step.
 The instructions from line 8 to line 13 of Algorithm 1 are repeated for all constraints in Γ^S. Within the loop, $\mathscr{A}$ and $\mathscr{E}_{\text{Reg}}$ operations require $O(1)$,
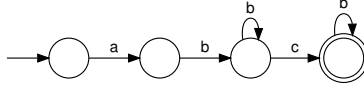
Figure 6: Product automaton of *Init*(a), *AlternateResponse*(a, b), *ChainPrecedence*(b, c), and *AtMostOne*(c)

since they are applied to a single constraint. To separate the sources of complexity inherent to the algorithm, and those coming from the manipulation of automata, we explicitly denote the complexity of the language-inclusion check of line 11 and the automata-product of line 12 as $\circlearrowleft_{\mathscr{L}_\supset}^{\mathrm{T}}$ and $\circlearrowleft_{A_\times}^{\mathrm{T}}$ respectively. With this notation at hand, we get that the loop from line 8 to line 13 of Algorithm 1 can be executed in $O(n \cdot (\circlearrowleft_{\mathscr{L}_\supset}^{\mathrm{T}} + \circlearrowleft_{A_\times}^{\mathrm{T}}))$.

At line 15 of Algorithm 1, a loop over the constraints in Γ^{U} starts. For every constraint in Γ^{U}, procedure *resolveConflictAndRedundancy* (Algorithm 2) is invoked. The time complexity of the procedure is affected again by the cross-product operation and the language-check. The recursive calls of lines 10, 12 and 13 are limited and independent of the number of constraints in Γ: The worst case is represented by a *ChainSuccession*(a, b) constraint passed in input as C, because it presents a chain of 4 subsuming constraints (hence, in 4 cases a $\mathscr{R}(C)$ exists) and has both a forward- and a backward-unidirectional relation constraint, namely *ChainResponse*(a, b) and *ChainPrecedence*(a, b). For each of them, 4 further subsuming constraints exist as well (cf. Figure 1(b)). The total amount of recursive calls is thus at most 12. However, every invocation of procedure *resolveConflictAndRedundancy* for each constraint can include in Γ^{R} up to 3 new constraints in place of the passed one, because the subsuming, forward and backward-unidirectional relation constraints may be added in line 7, executed within the respective recursive invocations at lines 10, 12 and 13. Therefore, $|\Gamma^{\mathrm{R}}| \leqslant 3 \cdot n$, and, in turn, the time required by the loop starting at line 15 of Algorithm 1 is $O(n \cdot (\circlearrowleft_{\mathscr{L}_\supset}^{\mathrm{T}} + \circlearrowleft_{A_\times}^{\mathrm{T}}))$.

Finally, procedure *resolveRedundancies*[II] (Algorithm 3) is invoked if and only if parameter r^{II} is set to $\top$ (line 17 of Algorithm 1). The constraints in Γ^{R} are preliminarily sorted at line 1 of Algorithm 3, with cost $O(n \cdot \log(n))$. Then, automaton A^{R} is built as the cross-product of all constraints in Γ^{R}. The computational complexity of this step is $O(n \cdot \circlearrowleft_{A_\times}^{\mathrm{T}})$. Within the loop starting at line 3, for every constraint $C \in \Gamma^{\mathrm{R}}$, the new automaton $A_{\cancel{C}}^{\mathrm{R}}$ is built as the cross-product of all constraints in Γ^{R} except C (line 5). Each

execution of the cross-product brings, again, a cost of $O(n \cdot \circlearrowleft_{A_\times}^{\mathrm{T}})$. At line 6, the languages of A^{R} and $A_{\mathscr{C}}^{\mathrm{R}}$ are compared, with cost $\circlearrowleft_{\mathscr{L}_\supset}^{\mathrm{T}}$. All in all, the computational cost of $resolveRedundancies^{\mathrm{II}}$ is thus $O(n^2 \cdot \circlearrowleft_{A_\times}^{\mathrm{T}} + n \cdot \circlearrowleft_{\mathscr{L}_\supset}^{\mathrm{T}})$.

To conclude, we notice that the two costs $\circlearrowleft_{A_\times}^{\mathrm{T}}$ and $\circlearrowleft_{\mathscr{L}_\supset}^{\mathrm{T}}$ can be uniformly represented by a single source of complexity $\circlearrowleft_{\mathscr{A}}^{\mathrm{T}}$, since the inclusion checks all depend on previous cross-product constructions.

Consequently, the overall algorithm runs in time:

$$O\Big(\underbrace{n \cdot \log n}_{\text{sorting}} + \underbrace{n \cdot \circlearrowleft_{\mathscr{A}}^{\mathrm{T}}}_{\text{conflict and redundancy (single) check}} + \underbrace{n^2 \cdot \circlearrowleft_{\mathscr{A}}^{\mathrm{T}}}_{\text{redundancy double-check}} \Big)$$

The three statements of the theorem directly follows, by noticing that computing the cross-product automaton of (at most) n regular expressions is, in the worst case, exponential in n [44, 45], and that the third element of the sum above is only present when r^{II} is true. $\qquad\qquad\square$

6. Experiments and Results

In this section, we illustrate the evaluation of our implemented approach. The approach has been validated in terms of *(i)* efficacy, measured by the number of pruned redundant constraints and detected inconsistencies, and *(ii)* efficiency, measured as computation time. Experiments have been conducted on process models discovered from real-world collections of event logs, provided by the IEEE Task Force on Process Mining on the 3TU Datacentrum platform,[1] i.e.,

- the event log of a loan application process of a Dutch financial institute, published in the context of the BPI challenge 2012 [43],
- the collection of three event logs of the Volvo IT incident and problem management, resp. describing closed issues, incidents, and open problems, from the BPI challenge 2013 [46],
- the event log of ITIL processes of Rabobank Group ICT, from the BPI challenge 2014 [47], and
- the event log of a road traffic fines management process [48].

All experiments were run on a machine equipped with an Intel Core i5-3320M, CPU at 2.60GHz, quad-core, Ubuntu Linux 12.04 operating

---

[1]`http://data.3tu.nl/repository/collection:event_logs_real`

Original model	306	cns.		Original model	69	cns.
Detected conflicts	2	cns.		Detected conflicts	0	cns.
Detected redundancies	174	cns.		Detected redundancies	28	cns.
Gain	57.52	%		Gain	40.58	%
Time	9,171	msec		Time	2,764	msec

(a) MINERful   (b) Declare Maps Miner

Table 3: Results of the application of the approach on the models discovered from the BPIC 2012 log [43] by MINERful and Declare Maps Miner.

system. The tool was implemented in Java SE 7 and integrated with the MINERful declarative process miner. It can be downloaded at: www.github.com/cdc08x/MINERful.

In the following subsections, we show *(i)* an in-depth analysis of the results obtained by checking consistency and redundancy of the models discovered with MINERful and Declare Maps Miner from the loan application process log and *(ii)* a summary of the results obtained from the redundancy check conducted over the models discovered from all aforementioned logs. We recall here that both MINERful and Declare Maps Miner already provide ad-hoc techniques to reduce the size of the returned models [28, 19]. Nevertheless, no mechanism allows them to remove inconsistencies or those non-trivial redundancies that our approach is able to find out.

6.1. Consistency and Redundancy Checking (BPI challenge 2012)

Here, we describe the outcome of the application of the proposed approach to detect inconsistencies over a declarative process model discovered from the event log provided for the BPI challenge 2012. The BPI challenge 2012 log was chosen for such an analysis because it is the one that presents inconsistencies in the discovered model already at relatively elevated thresholds of support. The event log pertains to an application process for personal loans or overdrafts. It contains 262,200 events distributed across 24 event classes and includes 13,087 traces. With this experiment, we show that our approach is capable of pruning the discovered models by detecting inconsistencies within the constraints discovered by two state-of-the-art declarative process discovery algorithms: MINERful and Declare Maps Miner. We set up both miners to return constraints with a support higher than 75%, a confidence higher than 12.5%, and an interest factor higher than 12.5%. In

a realistic scenario, indeed, event logs could contain errors due to recording mistakes or exceptional deviations from the usual execution [7]. Therefore, it makes sense to include those rules that are not fulfilled in the totality of the cases. Since we do not know how many errors or exceptional process enactments affected the log under analysis, we based our choice upon previous studies on the sensitivity of the discovered DECLARE constraints to the presence of noise in event logs [26]. The levels of confidence and interest factor are motivated by the need to limit the unavoidable increased number of constraints that are included in the result as the support threshold is lowered.

Table 3 summarises the results of the experiment. In the first set of experiments (Table 3(a)), we used MINERful. The number of discovered constraints was 306. On top of that, we applied the proposed algorithm, setting $\langle \leqslant_\leadsto, \leqslant_{\sigma\kappa\iota}, \leqslant_{\text{T}\sqsupseteq} \rangle$ as the ordering relation symbols and r^{II} to false. We obtained 130 constraints in total, with an execution time of 9,171 milliseconds. In the original set of 306, there were two sets of conflicting constraints that made the entire model inconsistent. The first set was $NotChainSuccession(\mathsf{A\_Preaccepted}, \mathsf{W\_Completeren\ aanvrag})$, $ChainResponse(\mathsf{A\_Preaccepted}, \mathsf{W\_Completeren\ aanvrag})$, and $Participation(\mathsf{A\_Preaccepted})$. The second set was $NotChainSuccession(\mathsf{W\_Completeren\ aanvraag}, \mathsf{A\_Accepted})$, $ChainResponse(\mathsf{W\_Completeren\ aanvraag}, \mathsf{A\_Accepted})$, and $Participation(\mathsf{W\_Completeren\ aanvraag})$. Both inconsistencies were detected by our algorithm. Note that the percentage of reduction over the set of discovered constraints (that was already pruned based on the subsumption hierarchy) was of 58%.

In the second set of experiments (Table 3(b)), we used the Declare Maps Miner. We discovered a set of constraints using the same thresholds for support, confidence and interest factor adopted for the previous experiment. The tool (that provides an ad-hoc technique for pruning) discovered 69 constraints. By applying the proposed algorithm starting from this set, we obtained 41 constraints (with an execution time of 2,764 milliseconds). The percentage of reduction was still around 40%.

Redundant constraints can be pruned based on complex reduction rules that are not supported by the state-of-the-art declarative process discovery algorithms. For example, from our experiments, we derived that $AtMostOne(\mathsf{A\_Finalized})$ becomes redundant due to the presence in combination of $AtMostOne(\mathsf{A\_PartlySubmitted})$, $Participation(\mathsf{A\_PartlySubmitted})$,

and $AlternatePrecedence$(A_PartlySubmitted, A_Finalized). Indeed, $Participation$(A_PartlySubmitted) and $AtMostOne$(A_PartlySubmitted) combined ensure that A_PartlySubmitted occurs exactly once. Then $AlternatePrecedence$(A_PartlySubmitted, A_Finalized) ensures that either A_Finalized does not occur or if it occurs it is preceded by the unique occurrence of A_PartlySubmitted without the possibilities of other occurrences of A_Finalized in between. Another example is $NotSuccession$(W_Nabellen offertes, A_Submitted), which is redundant with the combination of $Init$(A_Submitted), $AtMostOne$(A_PartlySubmitted), $Participation$(A_PartlySubmitted), and $ChainSuccession$(A_Submitted, A_PartlySubmitted). Indeed, $AtMostOne$(A_PartlySubmitted) and $Participation$(A_PartlySubmitted) combined ensure that A_PartlySubmitted occurs exactly once. This constraint in combination with $ChainSuccession$(A_Submitted, A_PartlySubmitted) and $Init$(A_Submitted) ensures that A_Submitted occurs only once at the beginning of every trace and, therefore, it can never occur after any other activity.

We want to finally remark that the pruning of redundancies of our approach does not alter the behaviour of the returned model. Those constraints that are eliminated are indeed those that can be removed without affecting the way in which the process can be enacted, because they do not restrict the possible executions any further. This is substantially different from the pruning based on thresholds like support, confidence and interest factor that modifies the set of the allowed behaviours. The next subsection illustrates a comprehensive view on the results of redundancy checking performed on a large set of real-world event logs.

6.2. Redundancy Checking Analysis

To assess the capability of the proposed approach to identify and prune the redundant constraints from a model, we ran the implemented algorithm on process models discovered from real-world logs. We utilised MINERful to discover the declarative process models from every log provided by the past editions of the BPI challenges in *(i)* 2012, *(ii)* 2013, *(iii)* 2014, and from *(iv)* the real-world event log of road traffic fines management process. Hereinafter, we will refer to the respective discovered process models as *(i)* "BPIC 2012", *(ii)* "BPIC 2013/1", "BPIC 2013/2", and "BPIC 2013/3", *(iii)* "BPIC 2014", and *(iv)* "Fines".

We have set the support threshold to 75% for the discovery phase. For the first log in the list, we have set the thresholds for confidence and interest factor to 25% and 12.5%, respectively, whereas for the remaining ones we

41

have used the thresholds 12.5% and 6.25%. These values were chosen so as to keep the returned constraints in a range that allowed several computationally intensive routines: *(i)* 226 for BPIC 2012, *(ii)* 30 for BPIC 2013/1, 76 for BPIC 2013/2, and 20 for BPIC 2013/3, *(iii)* 108 for BPIC 2014, and *(iv)* 46 for Fines. We have applied our technique on every discovered process model using every combination of defined ordering relations $\leqslant_{\rightsquigarrow}$, $\leqslant_{T\sqsupseteq}$, and $\leqslant_{\sigma\kappa\iota}$, plus a random sort used as a baseline. In all the aforementioned cases, the proposed algorithm was run twice: Once having the boolean flag r^{II} set to true, thus enabling the second pass over the pruned constraints and once having r^{II} set to false. For every run, we have measured the number of redundancies pruned, the computation time needed, and the average support σ, confidence κ, and interest factor ι of the returned constraints. We use the first metric to assess the efficacy of our approach, the second one to evaluate its efficiency, and the last three to estimate the fitness of the pruned model w.r.t. the original log.

Table 4 shows the obtained results on BPIC 2014 with (Table 4(b)) and without (Table 4(a)) the second-pass procedure enabled, respectively. The ordering relations are listed in the tables in their order of application. For the sake of readability, only the subscript under the $\geqslant$ symbol is shown: Hence, e.g., "T$\sqsupseteq$ $\rightsquigarrow$" stands for the consecutive application of ordering relations $\leqslant_{T\sqsupseteq}$ and $\leqslant_{\rightsquigarrow}$ to sort the constraints. In Table 4(a), the highlighted lines show the best sorting policies when the second pass is not enabled in terms of number of computation time, average support/confidence/interest factor of the returned constraints, and detected redundancies. The results confirm the influence of the adopted sequences of ordering relations on the examined metrics: The lowest computation time is achieved when $\leqslant_{\rightsquigarrow}$ is applied first (1,027 milliseconds), the highest combination of average σ, κ, ι is obtained when $\leqslant_{\sigma\kappa\iota}$ is applied first (resp., 0.932, 0.473, 0.369), and the highest number of detected redundancies (30 over 108, i.e., 27.778%) is obtained when $\leqslant_{T\sqsupseteq}$ is applied first. Table 4(b) shows the effect of the application of the second-pass check: The number of detected inconsistencies considerably raises in the range of 38 to 40 with a tangible gain of 26% to 166.667% over the first pass. This, however, comes at the price of a far slower computation time, about 10 to 20 times slower, and of a general decrease in terms of average support/confidence/interest factor. We remark here that the number of performed checks remains acceptable despite the second pass: In no case they amount to more than 199, hence not more than twice the number of the constraints in the original model (108). This helps the computation time to

	Sorting		Redundancies	Time [msec]	Avg. Supp.%	Avg. Conf.%	Avg. IntF.%
⇝	T⊒	σκι	28	1154	92.683	46.402	37.494
⇝	σκι	T⊒	27	**1027**	92.774	46.218	37.420
T⊒	⇝	σκι	29	1911	92.583	45.919	37.112
T⊒	σκι	⇝	28	1677	92.488	45.831	36.965
σκι	⇝	T⊒	15	1584	93.223	47.315	36.936
σκι	T⊒	⇝	15	1737	**93.223**	**47.315**	**36.936**
⇝	σκι		27	1064	92.774	46.218	37.420
⇝	T⊒		28	1078	92.683	46.402	37.494
T⊒	σκι		28	1629	92.488	45.831	36.965
T⊒	⇝		**30**	1899	92.488	45.444	36.525
σκι	T⊒		15	1731	93.223	47.315	36.936
σκι	⇝		15	1608	93.223	47.315	36.936
⇝			28	1231	92.683	46.402	37.494
T⊒			30	1565	92.488	45.444	36.525
σκι			15	1824	93.223	47.315	36.936
Random			26	1451	92.273	46.493	36.927

(a) Single pass

	Sorting		Checks	Redundancies (2nd)		Time (2nd) [msec]		Avg. Supp.%	Avg. Conf.%	Avg. IntF.%
⇝	T⊒	σκι	186	38	(10)	24767	(23699)	91.639	43.108	34.395
⇝	σκι	T⊒	187	38	(11)	25937	(24884)	91.639	43.108	34.395
T⊒	⇝	σκι	185	40	(11)	25684	(23960)	91.383	42.909	35.295
T⊒	σκι	⇝	186	40	(12)	28954	(27250)	91.383	42.909	35.295
σκι	⇝	T⊒	199	40	(25)	35621	(34073)	91.401	43.122	35.472
σκι	T⊒	⇝	199	40	(25)	36020	(34422)	91.401	43.122	35.472
⇝	σκι		187	38	(11)	27071	(25929)	91.639	43.108	34.395
⇝	T⊒		186	38	(10)	24596	(23590)	91.639	43.108	34.395
T⊒	σκι		186	40	(12)	27362	(25715)	91.383	42.909	35.295
T⊒	⇝		184	40	(10)	27257	(25623)	91.383	42.909	35.295
σκι	T⊒		199	40	(25)	37793	(36104)	91.401	43.122	35.472
σκι	⇝		199	40	(25)	37316	(35684)	91.401	43.122	35.472
⇝			186	38	(10)	25771	(24652)	91.639	43.108	34.395
T⊒			184	40	(10)	27259	(25776)	91.383	42.909	35.295
σκι			199	40	(25)	36956	(35188)	91.401	43.122	35.472
Random			188	40	(14)	25041	(23788)	91.394	42.891	35.244

(b) Double pass

Table 4: Results of the experiments over the model discovered from the BPIC 2014 [47] log.

Process	Input cns.	Redundancies		1st pass		2nd pass	
BPIC 2012	226	204	(90.265%)	154	(68.142%)	50	(22.124%)
BPIC 2013/1	30	12	(40%)	9	(30%)	3	(10%)
BPIC 2013/2	76	36	(47.368%)	21	(27.632%)	15	(19.737%)
BPIC 2013/3	20	5	(25%)	5	(25%)	0	(0%)
BPIC 2014	108	38	(35.185%)	28	(25.926%)	10	(9.259%)
Fines	46	30	(65.217%)	25	(54.348%)	5	(10.87%)

Table 5: Highest amounts of detected redundancies in the process models.

remain under 40 seconds in all cases. We recall here that an exhaustive search would have required up to approximately 3×10^{32} checks over an equivalent number of cross-products between automata, thus being computationally infeasible.

Table 5 shows the boost effect in terms of detected redundancies given by the second-pass strategy. The highest numbers in terms of pruned redundant constraints are depicted there. Noticeably, 90.265%, 65.217%, and 47.368% of constraints are classified as redundant for BPIC 2012, Fines, and BPIC 2013/2, respectively. This entails that a significant number of constraints could have been omitted from the returned models without altering the set allowed behaviours.

Figure 7 shows the proportion of pruned constraints over all analysed logs using the sorting that worked best in terms of redundancy detection, i.e., $\langle \leqslant_{\top\sqsupseteq}, \rightsquigarrow \rangle$. The abscissae indicate the types of templates. Triples of bars respectively represent the cumulative number of constraints that sum up *(i)* in all input models, *(ii)* after the first redundancy check, and *(iii)* after the second-pass redundancy check. Horizontal lines describe the average percentage of pruned constraints after the two redundancy check phases, resp. 50.4% and 64.03%. Relation constraints in particular tend to be more subject to redundancy, because more than half of them are pruned by the application of the proposed algorithm. This can be due to several factors. First, existence constraints can imply many relation constraints, as in the case, e.g., of *Participation*(a), implying that also *RespondedExistence*(b, a) holds true for all $b \in \mathcal{A} \setminus \{a\}$. Furthermore, non-chain relation constraints are also transitive: Therefore, if, e.g., *Response*(a, b) and *Response*(b, c) hold true, also *Response*(a, c) must hold true. Finally, relation templates create a hierarchical structure of subsumptions, thus parents along the hierarchy branches tend to be often pruned out: This is the case, e.g., when *ChainPrecedence*(a, b) holds true, thus mak-
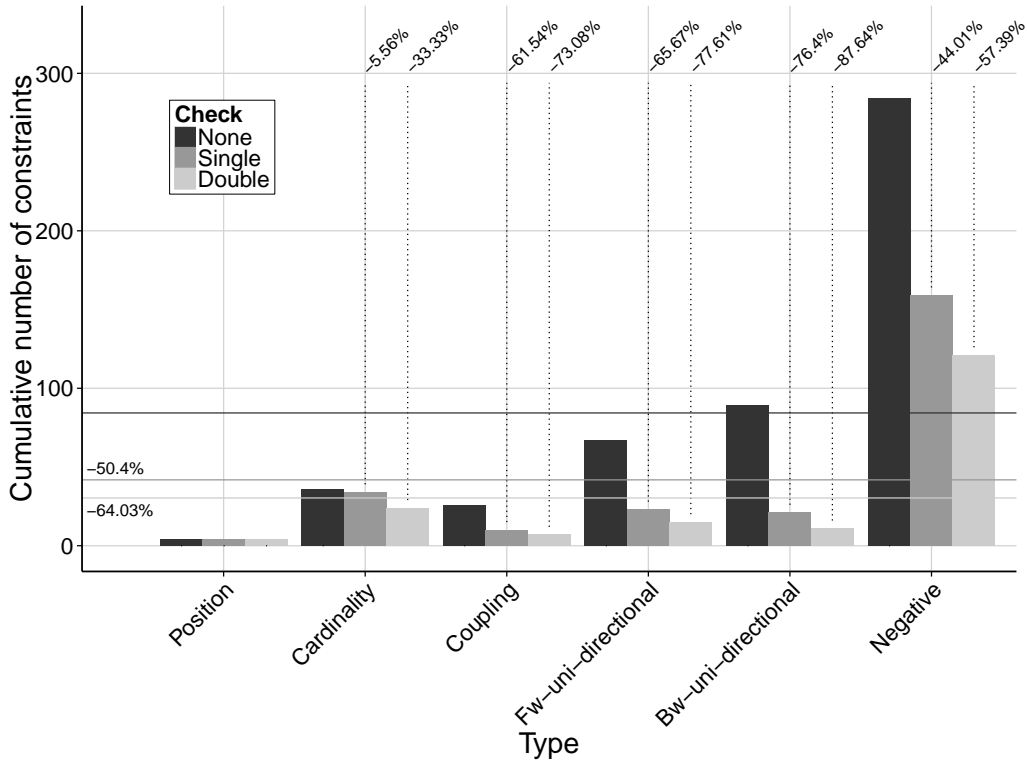
44

Figure 7: Redundancy reduction w.r.t. template types.

ing $AlternatePrecedence(\mathsf{a}, \mathsf{b})$, $Precedence(\mathsf{a}, \mathsf{b})$ and $RespondedExistence(\mathsf{b}, \mathsf{a})$ redundant.

Table 6 shows how the sorting order influences the resulting model in all the examined cases. All these results refer to the application of the proposed algorithm only in its first phase, because the second pass tend to alter and level off the metrics of interest. Table 6(a) presents the sequences of ordering relations that maximise the number of pruned constraints: For all models, $\leqslant_{\top\sqsubseteq}$ is the only ordering relation that occurs in every combination. Table 6(b) lists the sequences of ordering relations that allow for the highest support, confidence and interest factor. As expected, $\leqslant_{\sigma\kappa\iota}$ is always involved. Table 6(c) presents the best achieved computation times. In this case, $\leqslant_{\rightsquigarrow}$ is always in the list of ordering relations. Finally, Table 6(d) shows the application of a random sort as a baseline: As expected, the random sort leads to less efficacy in terms of detected redundancies, to a higher computation

Process	Sorting			Redundancies	Time [msec]	Avg. Supp.%	Avg. Conf.%	Avg. IntF.%
BPIC 2012	T⊒			172	2265	98.567	53.430	27.920
BPIC 2013/1	T⊒	σκι	⤳	9	262	92.659	37.463	29.696
BPIC 2013/2	T⊒	⤳	σκι	21	1165	96.129	36.108	25.423
BPIC 2013/3	⤳	σκι	T⊒	5	255	92.837	28.452	18.940
BPIC 2014	T⊒			30	1565	92.488	45.444	36.525
Fines	T⊒			25	328	96.744	54.648	33.710

(a) Maximimsing pruned redundancies

Process	Sorting			Redundancies	Time [msec]	Avg. Supp.%	Avg. Conf.%	Avg. IntF.%
BPIC 2012	⤳	σκι		138	3283	98.659	53.304	29.626
BPIC 2013/1	⤳	σκι	T⊒	6	287	93.384	42.105	35.299
BPIC 2013/2	σκι	⤳	T⊒	6	1376	96.546	39.187	28.764
BPIC 2013/3	σκι	T⊒	⤳	3	211	93.489	33.155	20.270
BPIC 2014	⤳	T⊒	σκι	28	1154	92.683	46.402	37.494
Fines	σκι	⤳		23	315	97.027	63.272	37.685

(b) Maximising support, confidence, and interest factor

Process	Sorting			Redundancies	Time [msec]	Avg. Supp.%	Avg. Conf.%	Avg. IntF.%
BPIC 2012	⤳			149	1643	99.163	52.332	29.551
BPIC 2013/1	σκι	⤳	T⊒	6	259	93.403	39.496	32.711
BPIC 2013/2	T⊒	⤳	σκι	21	1165	96.129	36.108	25.423
BPIC 2013/3	⤳	T⊒	σκι	3	210	93.489	33.155	20.270
BPIC 2014	⤳	σκι	T⊒	27	1027	92.774	46.218	37.420
Fines	σκι	⤳		23	315	97.027	63.272	37.685

(c) Minimising computation time

Process	Sorting	Redundancies	Time [msec]	Avg. Supp.%	Avg. Conf.%	Avg. IntF.%
BPIC 2012	Random	129	6041	97.475	60.052	29.940
BPIC 2013/1	Random	3	339	91.437	36.635	29.517
BPIC 2013/2	Random	11	1511	95.054	37.062	25.049
BPIC 2013/3	Random	0	265	92.936	28.043	19.053
BPIC 2014	Random	26	1451	91.394	42.891	35.244
Fines	Random	12	470	92.273	46.493	36.927

(d) Random

Table 6: Best set-ups for tests over process models discovered from real-world logs.

time, and to an average lower support, with respect to all the values achieved by the other sorting criteria.

The results listed in Table 7 are aggregated on the basis of applied ordering relations. In particular, each of them shows mean and standard deviation values for the metrics under analysis, resp. *(i)* Table 7(a) for the number of pruned constraints, *(ii)* Table 7(c) for the support, confidence and interest factor of the returned constraints, and *(iii)* Table 7(b) for the computation time. Highlighted rows evidence the set-ups performing best, i.e., *(i)* $\langle \leqslant_{T\sqsubseteq}, \leqslant_{\rightsquigarrow} \rangle$, allowing for pruning 42.5 constraints on average, *(ii)* $\langle \leqslant_{\rightsquigarrow} \rangle$, allowing for computation times of 854.33 milliseconds on average, *(iii)* $\langle \leqslant_{\sigma\kappa\iota}, \leqslant_{\rightsquigarrow} \rangle$, producing constraints that have an average support, confidence, and interest factor of resp. 95.331, 45.618, and 30.485.

The outcomes illustrated both in Tables 6 and 7 show that the user's choice on the sorting criteria influences the quality of the result in terms of (1) computation time, (2) pruned redundancies, or (3) fitness w.r.t. the event log. (1) When a faster computation is required, the order on the degree of activation linkage should be chosen. This is due to the fact that such a criterion speeds up the building of the product automaton, which is the most expensive operation in terms of computation time. The activations of the constraints with a higher degree of activation linkage are indeed involved as activations of several constraints. Consequently, the higher number of restrictions exerted on the same activation tends to be reflected in a limited number of states and transitions of the associated product-automaton. (2) To maximise the amount of pruned constraints, the partial order on the type and subsumption turns out to be the best choice. Such a criterion was indeed introduced for this purpose. The partial order on the type of constraints tends to rank as first those constraints that entail several other constraints, i.e., those that induce more redundancies. (3) Finally, to keep the constraints with a higher fitness w.r.t. the event log, the order on support, confidence, and interest factor of constraints should be taken into account, because it ranks first those constraints that were fulfilled and activated more often in the event log. As they are ranked first, they are assigned a higher priority when it comes to pruning redundancies out. Because the sorting criteria can be sequentially composed to build a strict total order over the constraints based on the hierarchical application of (partial) orders, different combinations of the aforementioned criteria can be used. The relatively small amount of required running time benefits an interactive selection of the criteria by the users. To further refine the results achieved, the second pass can be enabled

47

Sorting			Redundancies Mean	Std. Dev
$\leadsto$			35.333	56.330
$\leadsto$	T$\sqsupseteq$		36.333	58.267
$\leadsto$	T$\sqsupseteq$	$\sigma\kappa\iota$	34.667	52.796
$\leadsto$	$\sigma\kappa\iota$		33.667	51.717
$\leadsto$	$\sigma\kappa\iota$	T$\sqsupseteq$	34.167	52.442
T$\sqsupseteq$			42.500	64.214
T$\sqsupseteq$	$\leadsto$		42.500	64.190
T$\sqsupseteq$	$\leadsto$	$\sigma\kappa\iota$	41.000	59.097
T$\sqsupseteq$	$\sigma\kappa\iota$		39.500	58.298
T$\sqsupseteq$	$\sigma\kappa\iota$	$\leadsto$	39.333	57.895
$\sigma\kappa\iota$			31.500	51.725
$\sigma\kappa\iota$	$\leadsto$		31.333	51.321
$\sigma\kappa\iota$	$\leadsto$	T$\sqsupseteq$	31.667	52.129
$\sigma\kappa\iota$	T$\sqsupseteq$		31.833	52.533
$\sigma\kappa\iota$	T$\sqsupseteq$	$\leadsto$	31.833	52.533

(a) Detected redundancies

Sorting			Time [msec] Mean	Std. Dev
$\leadsto$			854.33	581.46
$\leadsto$	T$\sqsupseteq$		879.33	687.46
$\leadsto$	T$\sqsupseteq$	$\sigma\kappa\iota$	1009.17	993.07
$\leadsto$	$\sigma\kappa\iota$		1096.00	1156.43
$\leadsto$	$\sigma\kappa\iota$	T$\sqsupseteq$	1013.67	1028.96
T$\sqsupseteq$			1005.00	833.79
T$\sqsupseteq$	$\leadsto$		1077.83	875.61
T$\sqsupseteq$	$\leadsto$	$\sigma\kappa\iota$	1071.17	949.71
T$\sqsupseteq$	$\sigma\kappa\iota$		1121.17	1025.05
T$\sqsupseteq$	$\sigma\kappa\iota$	$\leadsto$	1147.50	1099.41
$\sigma\kappa\iota$			1214.83	1150.95
$\sigma\kappa\iota$	$\leadsto$		1204.50	1200.64
$\sigma\kappa\iota$	$\leadsto$	T$\sqsupseteq$	1211.00	1256.90
$\sigma\kappa\iota$	T$\sqsupseteq$		1170.33	1062.78
$\sigma\kappa\iota$	T$\sqsupseteq$	$\leadsto$	1186.17	1120.59

(b) Computation time

Sorting			Avg. Supp.% Mean	Std. Dev.	Avg. Conf.% Mean	Std. Dev.	Avg. IntF.% Mean	Std. Dev.
$\sigma\kappa\iota$			95.328	2.2175	45.527	10.7225	30.349	6.7563
$\sigma\kappa\iota$	$\leadsto$		95.331	2.2225	45.618	10.7773	30.485	6.6520
$\sigma\kappa\iota$	$\leadsto$	T$\sqsupseteq$	95.324	2.2123	45.551	10.7363	30.370	6.7394
$\sigma\kappa\iota$	T$\sqsupseteq$		95.321	2.2071	45.574	10.7506	30.391	6.7226
$\sigma\kappa\iota$	T$\sqsupseteq$	$\leadsto$	95.321	2.2071	45.574	10.7506	30.391	6.7226
$\leadsto$			95.477	2.7512	43.449	7.8523	30.041	5.9047
$\leadsto$	T$\sqsupseteq$		95.458	2.7320	43.455	7.9847	29.941	5.9725
$\leadsto$	T$\sqsupseteq$	$\sigma\kappa\iota$	95.365	2.5993	43.441	7.8767	29.840	5.9915
$\leadsto$	$\sigma\kappa\iota$		95.350	2.6817	44.216	10.0872	30.823	6.8060
$\leadsto$	$\sigma\kappa\iota$	T$\sqsupseteq$	95.335	2.6621	44.170	10.0439	30.751	6.8626
T$\sqsupseteq$			95.026	2.5021	43.296	9.2921	28.874	5.8584
T$\sqsupseteq$	$\leadsto$		95.054	2.5248	43.442	9.4212	28.856	5.7861
T$\sqsupseteq$	$\leadsto$	$\sigma\kappa\iota$	94.962	2.3475	43.226	9.0047	28.774	5.9934
T$\sqsupseteq$	$\sigma\kappa\iota$		94.914	2.3141	43.280	8.6290	28.839	5.8918
T$\sqsupseteq$	$\sigma\kappa\iota$	$\leadsto$	94.919	2.3220	43.251	8.6005	28.829	5.8968

(c) Support, confidence, and interest factor

Table 7: Average metrics of the processed models.

at the price of a higher computational effort.

Throughout this section, the results of the application of our approach to real-world benchmark data have been reported and analysed in depth. Experimental evidence has shown that the proposed algorithm can find inconsistencies in the discovered declarative models, which is an unprecedented result in the literature. Furthermore, redundancies are found within the discovered constraint sets, which amounted to approximately one third on average with a single-pass checking, and increased by a supplementary 15% with a second pass. Both achievements were yielded in reasonable computation times: Up to 3 seconds for the single-pass and up to 40 seconds for the double-pass. As per the experimental results, the proposed approach enhances the output of both MINERful and Declare Maps Miner, i.e., the two state-of-the-art declarative process miners. The following section examines the works published so far in the literature that deal with topics related to our research endeavour.

7. Related Work

Our research relates to three streams of research: Consistency checking for knowledge bases, research on process mining, and specifically research on DECLARE. Research in the area of knowledge representation has investigated the issue of consistency checking. In particular, in the context of knowledge-based configuration systems, Felfernig et al. [49] have challenged the problem of finding the core cause of inconsistencies within the knowledge base during its update test in terms of minimal conflicting sets (the so-called diagnosis). The proposed solution relies on the recursive partitioning of the (extended) constraint satisfaction problem into subproblems, skipping those that do not contain an element of the propagation-specific conflict [50]. In the same research context, the work described in [51] focuses on the detection of non-redundant constraint sets. The approach is again based on a divide-and-conquer approach, which favours, however, those constraints that are ranked higher in a lexicographical order. Differently from such works, we tend to exploit the characteristics of DECLARE templates in a sequential exploration of possible solutions. As in their proposed solutions, though, we base upon a preference-oriented ranking when deciding which constraints to keep in the returned set.

The idea to apply process mining in the context of workflow management systems has been introduced in [52]. Processes are modelled as directed

graphs in which vertices represent the activities and edges stand for the dependencies between them. Cook and Wolf [53], at the same time, investigate similar issues in the context of software engineering processes. They describe three methods for process discovery: *(i)* Neural network-based, *(ii)* purely algorithmic, and *(iii)* adopting a Markovian approach. The purely algorithmic approach builds a finite state machine where states are fused if their *futures* (in terms of possible behaviours for the next k steps) are identical. The Markovian approach uses a mixture of algorithmic and statistical methods, so as to cope with noise. However, the results presented in [53] are limited to sequential behaviour only. From [52] onwards, many techniques have been proposed. The α-algorithm [54] and its extensions α^{++} [55], $\alpha^{\#}$ [56], and $\alpha^{\$}$ [57] are algorithmic solutions that exploit behavioural relations between pairs of activities to discover a procedural process model from an event log. Differently from the declarative constraints of DECLARE, such relations are mutually exclusive: Since they do not overlap, they are by construction non-redundant. Behavioural profiles have been introduced by Weidlich et al. [58] as metrics to compare the similarity of procedural process models, as well as to measure the compliance of reported process executions w.r.t. a normative process model [59]. They also partition the product space of activities without semantic overlaps. The work of Polyvyanyy et al. [22] proposes a repertoire of exclusive behavioural relations between pairs of activities, along with a thorough study of their logical and mathematical properties. These behavioural relations are used as a means to analyse or discover procedural models, and cannot be applied to declarative languages.

Our work is related to research on declarative process discovery and modelling. In [9], the authors introduce the first version of Declare Maps Miner, an approach based on the instantiation of a set of candidate DECLARE constraints that are checked against an event log to identify the ones that are satisfied in a higher percentage of traces. This approach has been improved in [19] by reducing the number of candidates to be checked through an Apriori algorithm, originally developed by Agrawal and Srikant for mining association rules [60]. In [61], the same approach has been applied for the repair of DECLARE models based on log and for guiding the discovery task based on Apriori knowledge provided in different forms. In this work, some simple reduction rules are presented. These reduction rules are, however, not sufficient to detect redundancies due to complex interactions among constraints in a discovered model as demonstrated in our experimentation. In [10, 29], the authors present an approach for the mining of declarative process models

expressed through a probabilistic logic. The approach first extracts a set of integrity constraints from a log. Then, the learned constraints are translated into Markov Logic formulae that allow for a probabilistic classification of the traces. In [62, 20], the authors present an approach based on Inductive Logic Programming techniques to discover DECLARE process models. These approaches are not equipped with techniques for the analysis of the discovered models like the one presented in this paper. In [21, 28], the authors introduce MINERful, a two-step algorithm for the discovery of DECLARE constraints. As a first step, a knowledge base is built, with information about temporal statistics gathered from logs. Then, the statistical support of constraints is computed by querying that knowledge base. Also these works introduce a basic way to deal with redundancy based on the subsumption hierarchy of DECLARE templates that is non capable to deal with redundancies due to complex interactions of constraints. In [63], the authors propose an extension of the approach presented in [21, 28] to discover target-branched DECLARE constraints, i.e., constraints in which the target parameter is replaced by a disjunction of actual tasks. Here, as well as redundancy reductions based on the subsumption hierarchy of DECLARE constraints, also different aspects of redundancy are taken into consideration that are characteristic of target-branched DECLARE, such as set-dominance.

Different logic-based approaches have been used to define the semantics of the DECLARE templates. In principle, they have been expressed by means of LTL formulae [64], as in [31, 32]. Their interpretation on finite traces with LTL_f has been later clarified by [33, 34]. In [20], $\mathcal{S}$CIFF integrity constraints have been used, based on abductive logic programming [35]. Building on the fact that LTL_f has the same expressive power as FOL over finite traces [65, 66], the works in [28, 34] describe DECLARE templates in such formal representation. In [67], DECLARE constraints are translated into equivalent Petri nets with weighted, reset and inhibitor arcs. In [41, 21], REs are used to define the semantics of the DECLARE repertoire. Since REs and Monadic Second Order Logic (MSO) over finite traces [34, 68] have equivalent expressiveness, REs have a higher expressive power than LTL_f and, as such, are a suitable language to include the formulation of DECLARE. This additional expressiveness is exploited in [69] to model DECLARE meta-constraints that account for compensations and contextual constraints. Interestingly, the techniques presented in this article can be seamlessly applied to such enriched models.

Dynamic Condition Response Graphs (DCR Graphs) [70] are a well-

known declarative process modelling language alternative to Declare. They are not directly discussed from the perspective of consistency and redundancy [71], but can benefit from our work due to their grounding in Büchi automata [72].

Recently, De Smedt et al. [73] have conducted extensive studies on the so-called "hidden dependencies" [74], i.e., on the generation of implicit constraints tying activities due to the interaction of other constraints explicitly defined in a process model. The work of De Smedt et al. has lead to an approach that automatically uncovers the hidden dependencies in order to improve the understandability of declarative models.

8. Conclusion

In this paper, we addressed the problems of eliminating redundant and inconsistent constraint sets that are potentially generated by declarative process mining tools. After providing a formal definition of declarative models, we have formalised the problem and discussed its intractability due to its inherent exponential complexity. Thereupon, we have described our solution based on the notion of automata-product monoid and devised the corresponding analysis algorithms. The evaluation based on our prototypical implementation demonstrates that constraint sets discovered with state-of-the-art declarative process miners can be further pruned such that the result is consistent and locally minimal.

Our approach always finds all the conflicting constraints in a declarative process model. Furthermore, it substantially reduces the size of the discovered models by removing those constraints that do not alter by any means the allowed behaviours. It detects an elevated number of redundancies in process models returned by state-of-the-art discovery algorithms, notwithstanding the fact that they have built-in ad-hoc procedures to circumvent the problem of redundancy.

The sorting criteria adopted to sequentially check the constraints play a crucial role in determining the quality to prioritise in the result. The partial order on the type and subsumption yields a higher number of detected redundancies. The order on support, confidence, and interest factor promotes those constraints that are satisfied the most and involving the most frequent tasks, hence raising the average support, confidence and interest factor of the constraints in the returned model. The order on the degree of activation linkage favours a more efficient computation in terms of time. By choosing

which criteria to adopt and in which sequence, the user can thus influence the outcome. Nevertheless, it is in our plans to relieve the human actors from this choice, so as to let them specify the preferred target quality to strive for and make the algorithm automatically find a local optimum by trying different combinations of the aforementioned sorting criteria. Such an extension would be backed by the relatively small computation time required to return the intermediate results to be compared. As shown in the paper, the approach is capable of processing real-world process models in relatively short time.

Furthermore, we are planning to integrate our approach with the one of De Smedt et al. [73], which unveils the hidden dependencies among DE-CLARE constraints. The latter would drive the search for redundancies seen as dependencies from a core set of constraints, interactively decided by the user. It is also in our plans to involve the users to receive feedback on the outcome of the pruning technique in order to gain a better understanding on the perceived quality of the results, in a similar way to the studies reported in [75, 28].

In future research, we also aim at extending our work towards redundancy freedom, so as to ensure that, although not necessarily minimal, the discovered model is provably free of redundancies. Furthermore, we want to go beyond the pure control-flow perspective and extend our technique to the case of data- and resource-aware declarative process mining. When mining declarative constraints with references to data and resources, one of the challenges is to identify comparable notions of subsumption and causes of inconsistency. We also plan to follow up on experimental research comparing Petri nets and DECLARE [1, 2]. Prior experiments in that regard shed light on the fact that the declarative modelling approach suffered from the lack of consistency and redundancy checks [76, 71]. The notions defined in this paper help design declarative and procedural process models that are equally consistent and minimal, such that an unbiased comparison would be feasible.

References

[1] D. Fahland, D. Lübke, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative versus imperative process modeling languages: The issue of understandability, in: T. A. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, R. Ukor

(Eds.), BMMDS/EMMSAD, Vol. 29 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 353–366. `doi:10.1007/978-3-642-01862-6_29`.

[2] D. Fahland, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative versus imperative process modeling languages: The issue of maintainability, in: S. Rinderle-Ma, S. W. Sadiq, F. Leymann (Eds.), Business Process Management Workshops, Vol. 43 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 477–488. `doi:10.1007/978-3-642-12186-9_45`.

[3] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541 –580. `doi:10.1109/5.24143`.

[4] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, Ensuring model consistency in declarative process discovery, in: H. R. Motahari-Nezhad, J. Recker, M. Weidlich (Eds.), BPM, Vol. 9253 of Lecture Notes in Computer Science, Springer, 2015, pp. 144–159. `doi:10.1007/978-3-319-23063-4_9`.
URL `http://dx.doi.org/10.1007/978-3-319-23063-4_9`

[5] M. Pesic, Constraint-based workflow management systems: Shifting control to users, Ph.D. thesis, Technische Universiteit Eindhoven (10 2008).
URL `http://repository.tue.nl/638413`

[6] W. M. P. van der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: Balancing between flexibility and support, Computer Science - R&D 23 (2) (2009) 99–113. `doi:10.1007/s00450-009-0057-9`.
URL `http://dx.doi.org/10.1007/s00450-009-0057-9`

[7] C. W. Günther, W. M. P. van der Aalst, Fuzzy mining - adaptive process simplification based on multi-perspective metrics, in: Alonso et al. [77], pp. 328–343. `doi:10.1007/978-3-540-75183-0_24`.

[8] C. Di Ciccio, M. Mecella, Studies on the discovery of declarative control flows from error-prone data, in: R. Accorsi, P. Ceravolo, P. Cudré-Mauroux (Eds.), SIMPDA, Vol. 1027 of CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 31–45.
URL `http://ceur-ws.org/Vol-1027/paper3.pdf`

[9] F. M. Maggi, A. J. Mooij, W. M. P. van der Aalst, User-guided discovery of declarative process models, in: CIDM, IEEE, 2011, pp. 192–199.
URL http://dx.doi.org/10.1109/CIDM.2011.5949297

[10] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic logic-based process mining, in: W. Faber, N. Leone (Eds.), CILC, Vol. 598 of CEUR Workshop Proceedings, CEUR-WS.org, 2010.
URL http://ceur-ws.org/Vol-598/paper17.pdf

[11] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, W. M. P. van der Aalst, XES, XESame, and ProM 6, in: P. Soffer, E. Proper (Eds.), Information Systems Evolution - CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers, Vol. 72 of Lecture Notes in Business Information Processing, Springer, 2010, pp. 60–75. doi:10.1007/978-3-642-17722-4_5.

[12] T. Baier, C. Di Ciccio, J. Mendling, M. Weske, Matching of events and activities - an approach using declarative modeling constraints, in: K. Gaaloul, R. Schmidt, S. Nurcan, S. Guerreiro, Q. Ma (Eds.), BPMDS, Vol. 214 of Lecture Notes in Business Information Processing, Springer, 2015, pp. 119–134. doi:10.1007/978-3-319-19237-6_8.
URL http://dx.doi.org/10.1007/978-3-319-19237-6_8

[13] T. Baier, A. Rogge-Solti, J. Mendling, M. Weske, Matching of events and activities: an approach based on behavioral constraint satisfaction, in: R. L. Wainwright, J. M. Corchado, A. Bechini, J. Hong (Eds.), Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015, ACM, 2015, pp. 1225–1230. doi:10.1145/2695664.2699491.
URL http://doi.acm.org/10.1145/2695664.2699491

[14] W. M. P. van der Aalst, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer, 2011. doi:10.1007/978-3-642-19345-3.

[15] M. de Leoni, F. M. Maggi, W. M. P. van der Aalst, An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data, Information Systems 47 (2015) 258 – 277. doi:10.1016/j.is.2013.12.005.

[16] D. Fahland, W. M. P. van der Aalst, Model repair – aligning process models to reality, Information Systems 47 (2015) 220–243. `doi:10.1016/j.is.2013.12.007`.

[17] W. M. P. van der Aalst, H. A. Reijers, A. J. M. M. Weijters, B. F. van Dongen, A. K. Alves de Medeiros, M. Song, H. M. W. E. Verbeek, Business process mining: An industrial application, Inf. Syst. 32 (5) (2007) 713–732. `doi:10.1016/j.is.2006.05.003`.

[18] C. Di Ciccio, F. M. Maggi, J. Mendling, Efficient discovery of Target-Branched Declare constraints, Information Systems 56 (2016) 258 – 283. `doi:10.1016/j.is.2015.06.009`.
URL `http://www.sciencedirect.com/science/article/pii/S0306437915001271`

[19] F. M. Maggi, R. P. J. C. Bose, W. M. P. van der Aalst, Efficient discovery of understandable declarative process models from event logs, in: J. Ralyté, X. Franch, S. Brinkkemper, S. Wrycza (Eds.), CAiSE, Vol. 7328 of Lecture Notes in Computer Science, Springer, 2012, pp. 270–285.
URL `http://dx.doi.org/10.1007/978-3-642-31095-9_18`

[20] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Exploiting inductive logic programming techniques for declarative process mining, T. Petri Nets and Other Models of Concurrency 2 (2009) 278–295.
URL `http://dx.doi.org/10.1007/978-3-642-00899-3_16`

[21] C. Di Ciccio, M. Mecella, A two-step fast algorithm for the automated discovery of declarative workflows, in: CIDM, IEEE, 2013, pp. 135–142. `doi:10.1109/CIDM.2013.6597228`.
URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6588692`

[22] A. Polyvyanyy, M. Weidlich, R. Conforti, M. La Rosa, A. H. M. ter Hofstede, The 4c spectrum of fundamental behavioral relations for concurrent systems, in: G. Ciardo, E. Kindler (Eds.), Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings, Vol. 8489 of Lecture Notes in Computer Science, Springer, 2014, pp. 210–232.

doi:10.1007/978-3-319-07734-5_12.
URL http://dx.doi.org/10.1007/978-3-319-07734-5

[23] A. Polyvyanyy, A. Armas-Cervantes, M. Dumas, L. García-Bañuelos, On the expressive power of behavioral profiles, Formal Asp. Comput. 28 (4) (2016) 597–613. doi:10.1007/s00165-016-0372-4.

[24] C. Di Ciccio, M. Mecella, Mining constraints for artful processes, in: W. Abramowicz, D. Kriksciuniene, V. Sakalauskas (Eds.), BIS, Vol. 117 of Lecture Notes in Business Information Processing, Springer, 2012, pp. 11–23. doi:10.1007/978-3-642-30359-3_2.
URL http://dx.doi.org/10.1007/978-3-642-30359-3

[25] M. Räim, C. Di Ciccio, F. M. Maggi, M. Mecella, J. Mendling, Log-based understanding of business processes through temporal logic query checking, in: R. Meersman, H. Panetto, T. S. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, T. Sellis (Eds.), CoopIS, Vol. 8841 of Lecture Notes in Computer Science, Springer, 2014, pp. 75–92. doi:10.1007/978-3-662-45563-0_5.
URL http://dx.doi.org/10.1007/978-3-662-45563-0

[26] C. Di Ciccio, M. Mecella, J. Mendling, The effect of noise on mined declarative constraints, in: P. Ceravolo, R. Accorsi, P. Cudre-Mauroux (Eds.), Data-Driven Process Discovery and Analysis, Vol. 203 of Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2015, pp. 1–24. doi:10.1007/978-3-662-46436-6_1.
URL http://dx.doi.org/10.1007/978-3-662-46436-6_1

[27] C. C. Aggarwal, Data Mining - The Textbook, Springer, 2015. doi:10.1007/978-3-319-14142-8.
URL http://dx.doi.org/10.1007/978-3-319-14142-8

[28] C. Di Ciccio, M. Mecella, On the discovery of declarative control flows for artful processes, ACM Trans. Manage. Inf. Syst. 5 (4) (2015) 24:1–24:37. doi:10.1145/2629447.

[29] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic declarative process mining, in: Y. Bi, M.-A. Williams (Eds.), KSEM, Vol. 6291 of Lecture Notes in Computer Science, Springer, 2010, pp. 292–303. doi:

10.1007/978-3-642-15280-1_28.
URL http://dx.doi.org/10.1007/978-3-642-15280-1_28

[30] D. M. M. Schunselaar, F. M. Maggi, N. Sidorova, Patterns for a log-based strengthening of declarative compliance models, in: J. Derrick, S. Gnesi, D. Latella, H. Treharne (Eds.), IFM, Vol. 7321 of Lecture Notes in Computer Science, Springer, 2012, pp. 327–342. doi:10.1007/978-3-642-30729-4_23.

[31] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, Declare: Full support for loosely-structured processes, in: EDOC, IEEE Computer Society, 2007, pp. 287–300.
URL http://doi.ieeecomputersociety.org/10.1109/EDOC.2007.25

[32] A. H. M. ter Hofstede, W. M. P. van der Aalst, M. Adamns, N. Russell (Eds.), Modern Business Process Automation: YAWL and its Support Environment, Springer, 2010.
URL http://www.springer.com/computer+science/database+management+%26+information+retrieval/book/978-3-642-03120-5

[33] G. De Giacomo, R. De Masellis, M. Montali, Reasoning on LTL on finite traces: Insensitivity to infiniteness, in: C. E. Brodley, P. Stone (Eds.), Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada., AAAI Press, 2014, pp. 1027–1033.
URL http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8575

[34] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: F. Rossi (Ed.), IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013, IJCAI/AAAI, 2013.
URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997

[35] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, P. Torroni, Verifiable agent interaction in abductive logic programming: The sciff

framework, ACM Trans. Comput. Log. 9 (4) (2008) 29:1–29:43. `doi:`
`10.1145/1380572.1380578`.

[36] S. Gisburg, G. F. Rose, Preservation of languages by transducers, Information and Control 9 (2) (1966) 153 – 176. `doi:DOI:10.1016/S0019-9958(66)90211-7`.
URL `http://www.sciencedirect.com/science/article/pii/S0019995866902117`

[37] R. McNaughton, H. Yamada, Regular expressions and state graphs for automata, IEEE Transactions on Electronic Computers EC-9 (1) (1960) 39 – 47. `doi:10.1109/TEC.1960.5221603`.

[38] N. Chomsky, G. A. Miller, Finite state languages, Information and Control 1 (2) (1958) 91–112.

[39] A. Church, Logic, arithmetic and automata, in: Proceedings of the international congress of mathematicians, 1962, pp. 23–35, about the correspondence of FSAs and RExs.

[40] M. O. Rabin, D. Scott, Finite automata and their decision problems, IBM J. Res. Dev. 3 (1959) 114–125. `doi:10.1147/rd.32.0114`.
URL `http://dx.doi.org/10.1147/rd.32.0114`

[41] J. Prescher, C. Di Ciccio, J. Mendling, From declarative processes to imperative models, in: R. Accorsi, P. Ceravolo, B. Russo (Eds.), SIM-PDA, Vol. 1293 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 162–173. `doi:10.13140/2.1.1577.4409`.
URL `http://ceur-ws.org/Vol-1293`

[42] F. M. Maggi, M. Westergaard, M. Montali, W. M. P. van der Aalst, Runtime verification of ltl-based declarative process models, in: S. Khurshid, K. Sen (Eds.), Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers, Vol. 7186 of Lecture Notes in Computer Science, Springer, 2011, pp. 131–146. `doi:10.1007/978-3-642-29860-8_11`.
URL `http://dx.doi.org/10.1007/978-3-642-29860-8_11`

[43] B. F. van Dongen, Real-life event logs – a loan application process, Second International Business Process Intelligence Challenge (BPIC'12) (2012). `doi:10.4121/uuid:`

3926db30-f712-4394-aebc-75976070e91f.
URL `http://dx.doi.org/10.4121/uuid:`
`3926db30-f712-4394-aebc-75976070e91f`

[44] S. Yu, Q. Zhuang, K. Salomaa, The state complexities of some basic operations on regular languages, Theor. Comput. Sci. 125 (2) (1994) 315–328. `doi:10.1016/0304-3975(92)00011-F`.

[45] S. Yu, State complexity of regular languages, Journal of Automata, Languages and Combinatorics 6 (2) (2001) 221.

[46] W. Steeman, Real-life event logs – an incident management process, Third International Business Process Intelligence Challenge (BPIC'13) (2013). `doi:10.4121/uuid:`
`a7ce5c55-03a7-4583-b855-98b86e1a2b07`.
URL `http://dx.doi.org/10.4121/uuid:`
`a7ce5c55-03a7-4583-b855-98b86e1a2b07`

[47] B. F. van Dongen, Real-life event logs – logs from itil processes of rabobank group ict, Fourth International Business Process Intelligence Challenge (BPIC'14) (2014). `doi:10.4121/uuid:`
`c3e5d162-0cfd-4bb0-bd82-af5268819c35`.

[48] M. de Leoni, F. Mannhardt, Road traffic fine management process (2015). `doi:10.4121/uuid:`
`270fd440-1057-4fb9-89a9-b699b47990f5`.

[49] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artif. Intell. 152 (2) (2004) 213–234. `doi:10.1016/S0004-3702(03)00117-6`.

[50] U. Junker, QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems, in: D. L. McGuinness, G. Ferguson (Eds.), Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA, AAAI Press / The MIT Press, 2004, pp. 167–172.
URL `http://www.aaai.org/Library/AAAI/2004/aaai04-027.php`

[51] A. Felfernig, C. Zehentner, P. Blazek, COREDIAG: eliminating redundancy in constraint sets, in: 22nd Intl. Workshop on Principles of Diagnosis (DX), Munich, Germany, 2011.

[52] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: H.-J. Schek, G. Alonso, F. Saltor, I. Ramos (Eds.), Advances in Database Technology – EDBT'98, Vol. 1377 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1998, pp. 467–483, 10.1007/BFb0101003.
URL http://dx.doi.org/10.1007/BFb0101003

[53] J. E. Cook, A. L. Wolf, Discovering models of software processes from event-based data, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 215–249. doi:10.1145/287000.287001.
URL http://doi.acm.org/10.1145/287000.287001

[54] W. M. P. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1128–1142.
URL http://csdl.computer.org/comp/trans/tk/2004/09/k1143abs.htm

[55] L. Wen, W. M. P. van der Aalst, J. Wang, J. Sun, Mining process models with non-free-choice constructs, Data Min. Knowl. Discov. 15 (2) (2007) 145–180.
URL http://dx.doi.org/10.1007/s10618-007-0065-y

[56] L. Wen, J. Wang, W. M. P. van der Aalst, B. Huang, J. Sun, Mining process models with prime invisible tasks, Data Knowl. Eng. 69 (10) (2010) 999–1021. doi:10.1016/j.datak.2010.06.001.
URL http://dx.doi.org/10.1016/j.datak.2010.06.001

[57] Q. Guo, L. Wen, J. Wang, Z. Yan, P. S. Yu, Mining invisible tasks in non-free-choice constructs, in: H. R. Motahari-Nezhad, J. Recker, M. Weidlich (Eds.), Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings, Vol. 9253 of Lecture Notes in Computer Science, Springer, 2015, pp. 109–125. doi:10.1007/978-3-319-23063-4_7.
URL http://dx.doi.org/10.1007/978-3-319-23063-4

[58] M. Weidlich, J. Mendling, M. Weske, Efficient consistency measurement based on behavioral profiles of process models, IEEE Trans. Software Eng. 37 (3) (2011) 410–429. `doi:10.1109/TSE.2010.96`.

[59] M. Weidlich, A. Polyvyanyy, N. Desai, J. Mendling, M. Weske, Process compliance analysis based on behavioural profiles, Inf. Syst. 36 (7) (2011) 1009–1025. `doi:10.1016/j.is.2011.04.002`.

[60] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: J. B. Bocca, M. Jarke, C. Zaniolo (Eds.), VLDB, Morgan Kaufmann, 1994, pp. 487–499.
URL `http://www.vldb.org/conf/1994/P487.PDF`

[61] F. M. Maggi, R. P. J. C. Bose, W. M. P. van der Aalst, A knowledge-based integrated approach for discovering and repairing declare maps, in: C. Salinesi, M. C. Norrie, O. Pastor (Eds.), CAiSE, Vol. 7908 of Lecture Notes in Computer Science, Springer, 2013, pp. 433–448. `doi:10.1007/978-3-642-38709-8_28`.

[62] E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Inducing declarative logic-based models from labeled traces, in: Alonso et al. [77], pp. 344–359. `doi:10.1007/978-3-540-75183-0_25`.
URL `http://dx.doi.org/10.1007/978-3-540-75183-0_25`

[63] C. Di Ciccio, F. M. Maggi, J. Mendling, Discovering Target-Branched Declare constraints, in: S. W. Sadiq, P. Soffer, H. Völzer (Eds.), BPM, Vol. 8659 of Lecture Notes in Computer Science, Springer, 2014, pp. 34–50. `doi:10.1007/978-3-319-10172-9_3`.

[64] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2001.

[65] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the temporal analysis of fairness, in: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '80, ACM, New York, NY, USA, 1980, pp. 163–173. `doi:10.1145/567446.567462`.

[66] V. Diekert, P. Gastin, First-order definable languages, in: J. Flum, E. Grädel, T. Wilke (Eds.), Logic and Automata, Vol. 2 of Texts in Logic and Games, Amsterdam University Press, 2008, pp. 261–306.

[67] J. De Smedt, S. K. Vanden Broucke, J. De Weerdt, J. Vanthienen, A full r/i-net construct lexicon for declare constraints, Available at SSRN 2572869 (2015). doi:10.2139/ssrn.2572869.
URL http://ssrn.com/abstract=2572869

[68] J. R. Büchi, Weak second-order arithmetic and finite automata, Mathematical Logic Quarterly 6 (1-6) (1960) 66–92, proof of MSO-FSA equivalence. doi:10.1002/malq.19600060105.

[69] G. De Giacomo, R. D. Masellis, M. Grasso, F. M. Maggi, M. Montali, Monitoring business metaconstraints based on LTL and LDL for finite traces, in: S. W. Sadiq, P. Soffer, H. Völzer (Eds.), Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings, Vol. 8659 of Lecture Notes in Computer Science, Springer, 2014, pp. 1–17. doi: 10.1007/978-3-319-10172-9_1.
URL http://dx.doi.org/10.1007/978-3-319-10172-9_1

[70] T. T. Hildebrandt, R. R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, in: K. Honda, A. Mycroft (Eds.), PLACES, Vol. 69 of EPTCS, 2010, pp. 59–73.
URL http://dx.doi.org/10.4204/EPTCS.69.5

[71] H. A. Reijers, T. Slaats, C. Stahl, Declarative modeling – an academic dream or the future for bpm?, in: F. Daniel, J. Wang, B. Weber (Eds.), Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings, Vol. 8094 of Lecture Notes in Computer Science, Springer, 2013, pp. 307–322. doi: 10.1007/978-3-642-40176-3_26.
URL http://dx.doi.org/10.1007/978-3-642-40176-3_26

[72] R. R. Mukkamala, T. T. Hildebrandt, From dynamic condition response structures to büchi automata, in: J. Liu, D. A. Peled, B. Wang, F. Wang (Eds.), 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2010, Taipei, Taiwan, 25-27 August 2010, IEEE Computer Society, 2010, pp. 187–190. doi:10.1109/TASE.2010.22.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5587707

[73] J. De Smedt, J. De Weerdt, E. Serral, J. Vanthienen, Improving understandability of declarative process models by revealing hidden dependencies, in: S. Nurcan, P. Soffer, M. Bajec, J. Eder (Eds.), CAiSE, Vol. 9694 of Lecture Notes in Computer Science, Springer, 2016, pp. 83–98. `doi:10.1007/978-3-319-39696-5_6`.
URL `http://dx.doi.org/10.1007/978-3-319-39696-5`

[74] C. Haisjackl, I. Barba, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, B. Weber, Understanding declare models: strategies, pitfalls, empirical results, Software & Systems Modeling (2014) 1–28`doi:10.1007/s10270-014-0435-z`.
URL `http://dx.doi.org/10.1007/s10270-014-0435-z`

[75] C. Di Ciccio, M. Mecella, Mining artful processes from knowledge workers' emails, IEEE Internet Computing 17 (5) (2013) 10–20. `doi:10.1109/MIC.2013.60`.

[76] S. Zugal, P. Soffer, J. Pinggera, B. Weber, Expressiveness and understandability considerations of hierarchy in declarative business process models, in: I. Bider, T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, S. Wrycza (Eds.), Enterprise, Business-Process and Information Systems Modeling, Vol. 113 of Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2012, pp. 167–181. `doi:10.1007/978-3-642-31072-0_12`.

[77] G. Alonso, P. Dadam, M. Rosemann (Eds.), Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings, Vol. 4714 of Lecture Notes in Computer Science, Springer, 2007.